UBFC
UNIVERSITÉ
BOURGOGNE FRANCHE-COMTÉ

UFC
UNIVERSITÉ
DE FRANCHE-COMTÉ

THÈSE DE DOCTORAT DE L'ÉTABLISSEMENT UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ

PRÉPARÉE À L'UNIVERSITÉ DE FRANCHE-COMTÉ

École doctorale n°37

Sciences Pour l'Ingénieur et Microtechniques

Doctorat d'Informatique

par

ANDRÉ NAZ

Distributed Algorithms for Large-Scale Robotic Ensembles: Centrality, Synchronization and Self-Reconfiguration

Thèse présentée et soutenue à Montbéliard, le 4 décembre 2017

Composition du Jury :

RÖMER KAY	Professeur des Universités à Graz University of Technology (Autriche)	Président
WATTENHOFER ROGER	Professeur des Universités à ETH Zurich (Suisse)	Rapporteur
CORRELL NIKOLAUS	Maître de conférences à University of Colorado of Boulder (Etats-Unis)	Examinateur
BOURGEOIS JULIEN	Professeur des Universités à l'Université Bourgogne Franche-Comté (France)	Directeur de thèse
GOLDSTEIN SETH COPEN	Maître de conférences à Carnegie Mellon University (Etats-Unis)	Co-directeur de thèse
PIRANDA BENOÎT	Maître de conférences à l'Université Bourgogne Franche-Comté (France)	Co-encadrant de thèse

N° 7 7 4 3 1

À ma famille
To my family

“Happiness lies in the joy of achievement
and the thrill of creative effort.”

FRANKLIN D. ROOSEVELT

ACKNOWLEDGMENTS

My special thanks go to my supervisors Benoît Piranda, Julien Bourgeois and Seth Copen Goldstein. The supervision and support they have provided to me over the past years has truly helped the progression of my thesis and made my PhD journey a memorable, educational and enjoyable experience.

I express my sincere gratitude to my thesis committee members who have reviewed my dissertation. Their feedback and ideas have been absolutely invaluable.

Furthermore, I would also like to thank Abderahman Ait-Ali, my dear and faithful friend who is currently a PhD student at the KTH Royal Institute of Technology in Stockholm, for his availability and his constructive comments on my thesis work.

I would also like to thank my new office mate and recently PhD student, Pierre Thalamy, for having re-read the first three chapters of this manuscript and provided useful comments. I also thank him for his engineering work on VisibleSim as an intern in 2016. His code refactoring and test suite helped me to save time and focus on the design of my algorithms and simulation experiments. I wish him all the best for his PhD!

I would like to thank Catherine Maire for her proofreading that greatly improved the overall quality of this manuscript.

I am also thankful to the administrative staff of the University of Franche-Comté and to that of Carnegie Mellon University, in particular to Sylvie Klinkas, Dominique Ménétrier, Murielle Figuière, Marie-Thérèse Barthelet and Christina Contreras for their administrative support during my thesis.

Moreover, I would like to take this opportunity to thank the National Agency for Research (ANR) for funding my PhD research. This work is indeed part of the ANR - CO^2Dim project (contracts ANR-12-IS02-0004-01).

In addition, I would like to thank my enthusiastic office mates, Nicolas Boillot, Thadeu Knychala Tucci and, more recently, Pierre Thalamy, along with all the laboratory colleagues with whom I spent pleasant moments. Thadeu will definitely remain for ever in my mind as an incredible table soccer player (or not ;)! All joking aside, I wish him all the best for the end of his PhD!

Mais avant tout, je tiens sincèrement à remercier ma famille et ma fiancée bien aimées pour leur soutien sans faille et leur patience sans limite durant mes années d'études. Je les remercie aussi pour leur relecture de mon manuscrit de thèse ainsi que leurs suggestions de corrections. Je remercie aussi tout particulièrement mes grands-parents d'être vaillamment venus assister à ma soutenance malgré leur grand âge et les nombreux kilomètres qui nous séparent. Je repense souvent aux cours de lecture de l'été 1996, chez mes grands-parents, pour combler mes lacunes à l'issue du cours préparatoire, aux nombreuses récitations de leçons le dimanche avec mes parents et aux coups de fil à ma tante pour discuter des cours d'histoire avant le brevet des collèges ! Que les années sont passées vite, merci à tous !

ABSTRACT

Technological advances, especially in the miniaturization of robotic devices foreshadow the emergence of large-scale ensembles of small-size resource-constrained robots that distributively cooperate to achieve complex tasks (e.g., modular self-reconfigurable robots, swarm robotic systems, distributed microelectromechanical systems, etc.). These ensembles are formed by independent, intelligent and communicating units which act as a whole ensemble. These units cooperatively self-organize themselves to achieve common goals. These systems are thought to be more versatile and more robust than conventional robotic systems while having at the same time a lower cost.

These ensembles form complex asynchronous distributed systems in which every unit is an embedded system with its own but limited capabilities. Coordination of such large-scale distributed embedded systems poses significant algorithmic issues and open new opportunities in distributed algorithms. In my thesis, I defend the idea that distributed algorithmic primitives suitable for the coordination of these ensembles should be both identified and designed.

In this work, we focus on a specific class of modular robotic systems, namely large-scale distributed modular robotic ensembles composed of resource-constrained modules that are organized in a lattice structure and which can only communicate with neighboring modules. We identified and implemented three building blocks, namely centrality-based leader election, time synchronization and self-reconfiguration.

We propose a collection of distributed algorithms to realize these primitives. We evaluate them using both hardware experiments and simulations on systems ranging from a dozen modules to more than ten thousand modules. We show that our algorithms scale well and are suitable for large-scale embedded distributed systems with scarce memory and computing resources.

Keywords: Distributed algorithms, Modular robotics, Centrality-based leader election, Time synchronization, Self-reconfiguration.

RÉSUMÉ

Les récentes avancées technologiques, en particulier dans le domaine de la miniaturisation de dispositifs robotiques, laissent présager l'émergence de grands ensembles distribués de petits robots qui coopéreront en vue d'accomplir des tâches complexes (e.g., robotique modulaire, robots en essaims, microsystèmes électromécaniques distribués). Ces grands ensembles seront composés d'entités indépendantes, intelligentes et communicantes qui agiront comme un ensemble à part entière. Pour cela, elles s'auto-organiseront et collaboreront en vue d'accomplir des tâches complexes. Ces systèmes présenteront les avantages d'être plus polyvalents et plus robustes que les systèmes robotiques conventionnels tout en affichant un prix réduit.

Ces ensembles formeront des systèmes distribués complexes dans lesquels chaque entité sera un système embarqué à part entière avec ses propres capacités et ressources toutefois limitées. Coordonner de tels systèmes pose des défis majeurs et ouvre de nouvelles opportunités dans l'algorithmique distribuée. Je défends la thèse qu'il faut d'ores et déjà identifier et implémenter des algorithmes distribués servant de primitives de base à la coordination de ces ensembles.

Dans ce travail, nous nous focalisons sur une classe particulière de robots, à savoir les robots modulaires distribués formant de grands ensembles de modules fortement contraints en ressources (mémoire, calculs, etc.), placés dans une grille régulière et capables de communiquer entre voisins connexes uniquement. J'ai identifié et implémenté trois primitives servant à la coordination de ces systèmes, à savoir l'élection d'un nœud central au réseau, la synchronisation temporelle ainsi que l'auto-reconfiguration.

Dans ce manuscrit, je propose un ensemble d'algorithmes distribués réalisant ces primitives. Les algorithmes développés dans le cadre de ce travail ont été évalués sur des modules matériels et par simulation avec des systèmes composés de quelques dizaines à plus d'une dizaine de milliers de modules. Ces expériences montrent que nos algorithmes passent à l'échelle et sont adaptés aux grands ensembles distribués de systèmes embarqués avec des ressources fortement limitées à la fois en mémoire et en calcul.

Mots-clés: algorithmique distribuée, robots modulaires, élection de leader basée sur la centralité, synchronisation temporelle, auto-reconfiguration.

Contents

List of Figures

List of Tables

List of Abbreviations

APSP: All-Pair Shortest Paths

BFS: Breadth-First Search

BFST: Breadth-First Spanning Tree

C2SR: Cylindrical-Catoms Self-Reconfiguration (algorithm)

MRTP: Modular Robot Time Synchronization Protocol

CW: Clockwise

CCW: Counter-Clockwise

IoT: Internet of Things

IoRT: Internet of Robotic Things

MSR: Modular Self-Reconfigurable Robot

LMR: Distributed modular robotic ensemble composed of resource-constrained identical modules that are organized in a lattice structure and communicate together using only neighbor-to-neighbor communications.

PC2LE: Probabilistic-Counter-based Central-Leader Election (algorithm)

PM: Programmable Matter

RTC: Real-Time Counter

SSSP: Single-Source Shortest Paths

UART: Universal Asynchronous Receivers/Transmitters

WSN: Wireless Sensor Network

1

INTRODUCTION

1.1/ PROBLEM STATEMENT

Before stating our research problems in detail, it seems to us necessary to expose in a few paragraphs our vision of the broad context and ecosystem in which this thesis takes place.

Since the invention of the computer and later on with the Internet, many human activities have been transferred into the digital world, or said differently, virtualized or cyberized [Ma et al., 2015]. Listening to music, reading a book or the news are now mostly digital activities. Social media have pushed this trend further by virtualizing every social link humans can have: friendly relationships with Facebook, professional links with LinkedIn, photos or multimedia content sharing with EyeEm, Instagram or Tumblr, to cite a very few. Computer gaming is now the principal leisure activity at nearly all ages and is now referred to as e-sport. Large-scale virtual worlds have been created and a new culture coming from e-sport has emerged, linking together people from all around the world as this culture is completely international and does not belong to any part of the world. For example, Leagues of Legends or Clash of Clans are two different games and universes with their own rules and a really international community. The next revolution will certainly be the democratization of virtual reality, augmented reality and mixed reality. Many companies have already marketed products in these emerging fields (e.g., Oculus VR with the Oculus Rift, Microsoft with HoloLens, Samsung, Magic Leap, etc.). This evolution could made us think that our lives will become virtual in great part.

Another trend exists. The cyber world is indeed getting into the physical one by embedding computation, communication and sensing capabilities into day-to-day products with the Internet of Things (IoT). Smart things that are nowadays deeply embedded in our daily life, are not only able to sense and react to their environment but also to interact with people and things, providing valuable services to human beings. The goal of the IoT is to “allow people and things to be connected anytime, anyplace, with anything and anyone, ideally using any path/network and any service” [Vermesan et al., 2013]. The IoT involves human-to-machine, machine-to-machine and human-to-human communications. Most popular IoT applications include smart homes, smart cities, smart environment, eHealth, smart supply chain, etc. For instance, smart homes propose a wide variety of domotic-related services (e.g., temperature management, user-friendly voice

assistant, intrusion detection, etc.). Smart cities and smart highways monitor the traffic in real time to optimize the driving experience (e.g., diversions according to traffic jams or climatic conditions). eHealth enables patients surveillance, fall detection of elderly people, etc. Smart behavior and cooperation among many interconnected smart devices rise significant algorithmic challenges. Smart objects are not only interconnected through the Internet but may also communicate together in an ad-hoc fashion, potentially forming large-scale infrastructure-less distributed systems.

Among these smart objects, robotic things go one step further by providing physical services as they can act over the physical world thanks to actuation capabilities. Intelligent and networked robotic devices form the Internet of Robotic Things (IoRT) [Vermesan et al., 2017]. Autonomous cars, collaborative floor-cleaning robots, co-working robots, etc. fall into the IoRT. Technological advances, especially in the miniaturization of robotic devices, foreshadow the emergence of large-scale ensembles of small-size robots that distributively cooperate to achieve complex tasks (e.g., modular robotic systems [Yim et al., 2009], swarm robotic systems [Şahin, 2004], distributed MicroElectroMechanical Systems (diMEMS) [Bourgeois et al., 2012], etc.). These ensembles are formed from independent, intelligent and communicating units which act as a whole ensemble. These units cooperatively self-organize in order to perform specific complex tasks and achieve common goals. These systems are thought to be more versatile and more robust than conventional robotic systems while having at the same time a lower cost. When considered as a whole ensemble, a set of such units is a full IoRT object that takes part in the IoT ecosystem. At the same time, when viewed as a set of interconnected units, the ensemble is a complex intranet of robotic things, in which every unit is an embedded system with its own but limited capabilities.

In line with that trend, this dissertation deals with distributed coordination in large-scale distributed embedded systems and more specifically in resource-constrained modular robotic ensembles. The Smart Blocks [Piranda et al., 2013] and the Claytronics [Goldstein et al., 2004] projects envision interesting applications based on large-scale modular robotic systems. The former aims to build a large distributed modular system to convey small and fragile objects, by attaching many modules together, each one equipped with a conveyance surface. The conveyance system can rearrange its global shape to self-adapt to new situations (e.g., new tasks, self-healing after a module failure, etc.). The goal of the Claytronics project is to use up to millions of micro modules to build Programmable Matter (PM), i.e., matter that can change its physical properties in response to external and programmed events.

PM is a physical instance of a virtual representation. This synthetic reality has a wide range of applications (e.g., sending/downloading copies of physical objects, morpheable objects reshapable at will, injectable surgical instruments, 3D interactive life-size TV, etc.). In addition, PM could also provide a bidirectional mapping between the virtual representation of an object and its physical one formed from PM. PM will, therefore, be a technology which will literally bridge the gap between the physical and the virtual worlds. It will enable people not only to control their environment but also to shape it. It goes even further as it will allow them to create intelligent objects that act like living things. Implications may drastically change our society. This concept definitely fits the ultimate desire of human

beings to master their world.

Modular robotics is a cross-disciplinary domain which poses both hardware and software challenges. In this thesis, we focus on software and algorithmic problems. With PM, people will, for instance, hold in their hands large-scale networks of resource-constrained micro robotic devices. Coordination of such large-scale distributed embedded systems poses significant algorithmic issues and opens new opportunities in distributed algorithms.

In my thesis, I defend the idea that distributed high-level algorithmic primitives suitable for the coordination of these ensembles should be both identified and designed. Besides, these needs were stated during the 2016 Dagstuhl Seminar on “Algorithmic Foundations of Programmable Matter” [Fekete et al., 2016]. This point of view has also been recently defended by Z. Derakhshandeh in her doctoral thesis entitled “Algorithmic Foundations of Self-Organizing Programmable Matter” [Derakhshandeh, 2017], in which she addresses the leader election, shape formation and coating problems in the theoretical Amoebot model [Derakhshandeh et al., 2014].

The complexity that lies in the coordination of these ensembles depends on the hardware features of the individual modules (computation power, communication model, structure organization, motion capabilities, etc.). Communication is central to module coordination. The communication model and the structure organization determine the overall network properties. Complexities of distributed algorithms are generally expressed as a function of network properties (e.g., number of nodes, number of links, node degree, radius/diameter of the system). Many algorithms target a specific class of networks. For instance, some algorithms are more efficient in sparse networks than in dense networks (e.g, the virtual coordinate-based routing protocol in [Zhao et al., 2007]). Thus, it is crucial to take into account the network properties in order to design and choose appropriate algorithms, especially in large-scale systems.

In this work, we focus on a specific class of modular robotic systems, namely distributed modular robotic ensembles composed of resource-constrained identical modules that are organized in a lattice structure and which can only communicate with neighboring modules. We name this class of robots LMRs. As explained in Section 2.2.4, this class captures a variety of existing systems and is particularly suitable to realize large-scale ensembles. In the same chapter, we show that these ensembles form asynchronous, sparse, low-degree, large-diameter and large-average-hop-distance networks.

The contributions of this thesis are motivated by the application scenario depicted in Figure 1.1. This scenario considers a Modular Self-Reconfigurable Robot (MSR) [Yim et al., 2007] composed of more than ten thousand millimeter-scale cylindrical rolling robots developed in the Claytronics project [Karagozler et al., 2009]. These robots move in 2D space and thus form 2D ensembles only. We call these robots the 2D Catoms, even if they are 3D objects. The term “catom” stands for “Claytronics atom”, which is the basic unit for Claytronics PM. In our scenario, the modular robot first self-reconfigures its shape into a car shape. Then, modules into the turn-signal area coordinate themselves in order to blink in a synchronized fashion. This scenario rises several challenges. We identified and addressed three of them, namely centrality-based leader election, time synchroniza-

tion and self-reconfiguration.

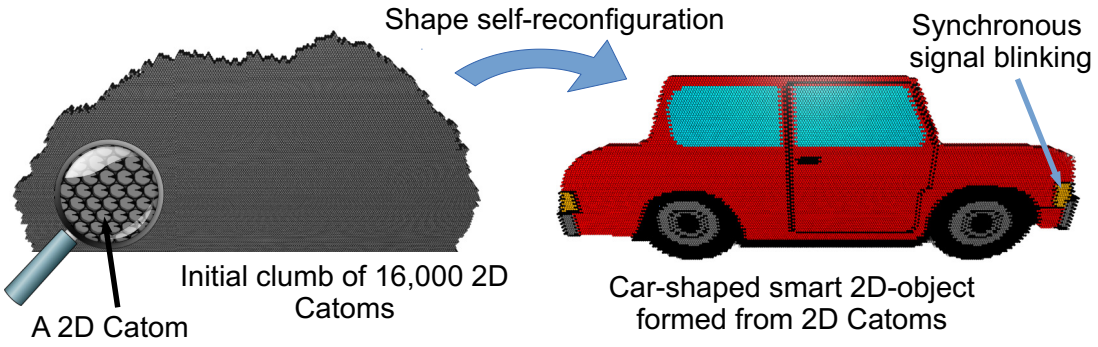


Figure 1.1: Application scenario that drives the research presented in this dissertation.[1]

The three algorithmic challenges we tackle in this thesis are:

- **Self-reconfiguration:** Self-reconfiguration is the process during which an MSR transforms itself from an initial configuration into a goal one. This process has several applications. In the context of programmable matter, it enables an MSR to assume different shapes. Self-reconfiguration can also be used to adapt an MSR to changes in the environment or to specific tasks. For instance, in [Lakhlef et al., 2013], the authors use self-reconfiguration to rearrange the modules connectivity in order to reach an optimal network topology. Self-reconfiguration poses several software challenges. Firstly, planning is challenging as the number of possible unique configurations is huge: $(c \times w)^n$ where n is the number of modules, c the number of possible connections per module and w the ways of connecting the modules together [Park et al., 2008]. Depending on the physical constraints, modules can often move concurrently, which makes the configuration space grow at the rate of $O(m^n)$ with m the number of possible movements and n the number of modules free to move [Barraquand et al., 1991]. The exploration space for reconfiguration between two random configurations is therefore exponential in the number of modules, which prevents us from finding a complete optimal planning for all but the simplest configurations. The optimal self-reconfiguration planning for chain-type MSRs is then an NP-complete problem [Hou et al., 2014], and, to the best of our knowledge, nothing has been proved so far for lattice-based MSRs. Secondly, in addition to the path planning problem, the self-reconfiguration process is also challenging as it is a distributed process that requires the distributed coordination of mobile autonomous modules connected in time-varying ways. In particular, modules have to coordinate their motions in order not to collide with each other. Self-reconfiguration algorithms are often tailored for a specific class of modular robots, with specific motion constraints. Here, we base our model on the 2D Catoms. Our research question in this perspective is: How to self-reconfigure an MSR composed of thousands of 2D

[1] The magnifying glass image is a modified version of an image taken from Pixabay https://pixabay.com. To generate the 2D-Catom car image, we used a modified version of a car image taken from Fotomelia http://www.fotomelia.com. Those two original images have been dedicated to the public domain under the Creative Commons CC0 license.

Catoms into various shapes?

- **Time Synchronization:** Coordination (e.g., synchronous blinking) among a group of modules often relies on the existence of a common notion of time. Every module has its own notion of time provided by its own hardware clock. Since common hardware clocks are imperfect, local clocks tend to run at slightly different and variable frequencies, drifting apart from each other over time. Consequently, a distributed time synchronization is necessary to keep the local clock of each module synchronized. Several approaches to time synchronization exist (continuous vs on-demand, network-wide vs clustering, timescale transformation vs clock synchronization, etc.) [Römer et al., 2005]. The approach to be used depends on the target application. In the continuous model, nodes strive to kept synchronized at all times. This model is opposed to the on-demand synchronization model where nodes can either a posteriori agree on the time at which an event has occurred or anticipate synchronizations in order to trigger some coordinated actions at a given time. In our application scenario, we aim at simultaneously and repeatedly executing a local algorithm, namely a color change. For this specific scenario, the existence of a common notion of time among all modules is required. Here, our goal is to achieve network-wide and continuous time synchronization. This is the most general approach. Synchronization protocols based on this approach aim to keep a small offset between local clocks and a global reference time. In most of the existing protocols, devices exchange timestamped messages in order to estimate the current global time. Since time keeps going during communications, modules have to correctly compensate for network delays in order to evaluate the current global time upon reception of synchronization messages. Although it is non-trivial to accurately estimate communication delays, especially in the presence of unpredictable delays (due for example, to queueing or retransmissions), it is crucial in order to achieve high-precision performance. In this work, we assume that every module is equipped with a local clock, which can be low-precision and low-resolution, typically in the order of the millisecond. Moreover, we target fairly static ensembles. Our research on this topic is driven by the questions: How to efficiently and accurately synchronize fairly-static large-scale distributed embedded ensembles in which entities are equipped with low-precision clocks and communicate with their immediate neighbors only? What is the largest network we can synchronize and how accurately?
- **Centrality-based leader election:** Many distributed algorithms require a specific role to be played by a leader, a single node in the system. The choice of this node often has a direct impact on the performance. Leaders are often used to provide such varied services as time synchronization, message routing [Blazevic et al., 2005], etc. In many algorithms and protocols, ensuring the proper selection of the leader is crucial for the performance. In particular, selecting a central node as the leader can significantly improve algorithm efficiency by reducing the network traffic or the time of convergence, especially in large-average-distance and large-diameter networks. For example, in time-master-based synchronization protocols, placing the time-master at a central node leads to more synchronization precision in large-diameter networks as the precision of remote clock readings tends to decrease with the hop distance (see Chapter 4). It is thus essential to have a fast and

efficient way to select a good leader. Several centrality definitions have been proposed in the literature. In this dissertation, we focus on the center and the centroid, i.e., the sets of nodes which respectively minimize the maximum and the average network distance to all the others. Classical distributed algorithms require global information about the connectivity network to elect a node that belongs to the exact center or centroid. Thus, they are not suitable for large-scale distributed embedded systems with scarce computation, memory and energy resources. Electing a central node actually involves a trade-off between the cost that can be afforded in terms of resources (time, memory, computation, energy) and the desired level of accuracy. This leads to the following research question: How to elect accurate approximate center and centroid nodes with both a reasonable convergence time and a limited memory usage in large-scale resource-constrained distributed embedded systems?

It must be well understood that we use the scenario presented in Figure 1.1 for illustrative purposes only. The primitives proposed in this thesis can be used to realize our scenario but we do not claim this is the only or the optimal way to do it. Moreover, this work is applicable to other applications and systems.

Although some functionalities of the 2D Catom have been physically validated by the realization of a hardware prototype (i.e., powering, adhesion and motion on a conductive surface) [Karagozler et al., 2009], no 2D-Catom ensemble has been erected yet and the current prototype still needs to be enhanced with different capabilities (e.g., communication). Hence, we use simulations in order to evaluate our algorithms on this platform. However, we consider that hardware deployment is an important step in the evaluation process of distributed algorithms. For experiments on hardware, we have at our disposal several dozen hardware Blinky Blocks [Kirby et al., 2011]. Blinky Blocks are centimeter-size modular robotic systems that were also developed in the Claytronics project. We evaluate compatible algorithms on this platform using both hardware experiments and simulations. Simulations enable evaluation in larger-scale ensembles. We present these two modular robotic systems in more detail in the next chapter.

We emphasize that our research strongly intersects with the work achieved in the fields of distributed systems, computer networks, sensor networks, ad-hoc networks, etc. In particular, centrality and time synchronization have been widely studied in the literature but rarely in ad-hoc networks composed of tens of thousands of resource-constrained devices. In this thesis, we address these problems from an efficiency and scalability perspective.

1.2/ CONTRIBUTIONS

In this thesis, we establish the network properties of our target systems and propose a collection of distributed algorithms to tackle our three research problems. It must be well understood that beyond proposing tailored contributions, our work is applicable to a variety of systems. We leverage the complete source code of all our algorithms on

GitHub[2,3].

The principal contributions of this thesis are:

- **Centrality-based leader-election algorithms:** We propose a collection of efficient and effective distributed algorithms to elect approximate-centroid and approximate-center nodes in asynchronous distributed systems. We introduce the k-BFS SumSweep framework, the ABC-Center algorithm and the Probabilistic-Counter-based Central-Leader Election (PC2LE) framework. Frameworks are declined in two versions, one for approximate-center node election, another for approximate-centroid node election. Our algorithms and frameworks do not require any prior knowledge of the network, have a well-defined termination criterion, converge in a reasonable amount of time and are memory-efficient. k-BFS SumSweep and ABC-Center perform distributed Breadth-First Search network traversals (BFSes) from a sample of nodes, while PC2LE uses probabilistic counting:
  - **k-BFS SumSweep:** In the k-BFS SumSweep, nodes compute their partial centrality value to a subset of root nodes composed of a random initial node and $k-1$ most external nodes. Root nodes are consecutively selected using the SumSweep approach that was originally proposed as a starting point of the sequential algorithms for exact radius and diameter computation of external graphs in [Borassi et al., 2014]. The main idea behind our framework is that central nodes are first and foremost central to the most external ones. Let n be the number of nodes in the system, m, the number of links and Δ the maximum network degree. Our framework runs in $O(kd)$ time using $O(mn^2)$ messages of size $O(1)$ and $O(\Delta)$ memory space per node[4]. As shown in the evaluation section, our framework provides good accuracy with small k values even in large-scale Blinky Blocks systems with more than 10^4 modules.
  - **ABC-Center:** ABC-Center[5] extends the sequential Minimax [Handler, 1973] and 4-Sweep [Crescenzi et al., 2013] algorithms. ABC-Center identifies an extreme path and recursively isolates midpoints on it until electing a single node. The main idea of ABC-Center is that central nodes lie in the middle of a diameter path. ABC-Center may be more convenient to use than the k-BFS framework as ABC-Center converges by itself, i.e., its termination does not rely on any input parameter. We propose two versions of ABC-Center. The latest version, ABC-CenterV2, runs in $O(sd)$ time using $O(mn^2)$ messages of size $O(1)$ and $O(\Delta)$ memory space per node, where s is the number of iterations that ABC-CenterV2 requires to terminate. ABC-Center requires only a few iterations in Blinky Blocks systems where nodes are organized in a simple-

[2]GitHub repository that hosts our algorithm codes for simulations: https://github.com/nazandre/thesis

[3]Official Blinky Blocks firmware repository in which some of our algorithm codes are hosted: https://github.com/claytronics/oldbb

[4]We adopt a system approach to quantify the asymptotic memory usage of our algorithms. Unless otherwise mentioned, memory complexities are expressed in machine words rather than in bits (see Section 3.2). The size of words, however, limits the number of nodes the network may contain. For instance, we assume that a node identifier is stored using a single word, thus, $O(1)$ memory space. If a word is composed of w bits, then the network may only contain up to 2^w nodes.

[5]Some examples of ABC-Center executions on Blinky Blocks systems are available online in video at https://youtu.be/QxK12UAq42o and https://youtu.be/PYnJn6tXKa8

cubic lattice.

  - **Probabilistic-Counter-based Central-Leader Election (PC2LE):** PC2LE is based on the input-graph analysis algorithms [Kang et al., 2011a, Kang et al., 2011b] and the distributed synchronous algorithm [Garin et al., 2012] which use low-memory-footprint probabilistic counters (e.g., Flajolet-Martin [Flajolet et al., 1985], HyperLogLog [Flajolet et al., 2007]) to estimate node centrality measures. In PC2LE, an estimated centrality value is computed for all nodes. PC2LE is approximately equivalent to running a BFS from every node but at less expense in terms of computations and communications. PC2LE runs in $O(d)$ time using $O(mn^2)$ messages of size $O(c)$ and $O(\Delta + c)$ memory space per node, where c is the memory complexity of the probabilistic counter that is used.

  To the best of our knowledge, our algorithms are the most precise existing distributed algorithms designed to elect an approximate centroid or an approximate center in our target systems, with both a reasonable convergence time and a limited storage cost.

- **Modular Robot Time Synchronization Protocol (MRTP)[6]:** We propose MRTP, a network-wide time synchronization protocol for modular robots with neighbor-to-neighbor communications. Our protocol achieves its performance by combining several mechanisms: central time-master election, selection of the most suited mechanism to compensate for communication delays depending on the target system and clock skew compensation using linear regression. MRTP is strongly inspired by time synchronization protocols proposed in ad-hoc wireless sensor networks (the Timing-sync Protocol for Sensor Networks (TPSN) [Ganeriwal et al., 2003], the Flooding Time Synchronization Protocol (FTSP) [Maróti et al., 2004] and the PulseSync protocol [Lenzen et al., 2009]). We evaluate our protocol on the Blinky Blocks system both on hardware and through simulations. Experimental results show that MRTP can potentially manage real systems composed of up to 27,775 Blinky Blocks. We show that our protocol is able to keep a Blinky Blocks system synchronized to a few milliseconds, using few network resources at runtime, even though the Blinky Blocks hardware clocks exhibit very poor accuracy and resolution. We compare MRTP to existing synchronization protocols ported to fit our system model. Simulation results show that MRTP exhibits on average a lower maximum pairwise synchronization error than the most precise compared protocols while sending more than half less messages in compact systems.
- **Cylindrical-Catoms Self-Reconfiguration (C2SR) algorithm[7]:** We propose C2SR, a self-reconfiguration algorithm for rolling cylindrical modules arranged in a two-dimensional vertical hexagonal lattice. Our algorithm is a parallel, asynchronous and decentralized distributed algorithm to self-reconfigure robots from an initial configuration into a goal one. It is able to manage almost any kind of initial and goal compact shapes (i.e., without any hole). Although our work is focused on

[6] Some examples of MRTP running on the Blinky Blocks platform are available online in video at https://youtu.be/66D12ESGc98 and https://youtu.be/X6QzivsmJBo

[7] Some examples of self-reconfiguration with C2SR are available online in video at https://youtu.be/XGnY-oS4Nw0

the algorithm, we carry out our analysis with respect to the hardware constraints of the 2D Catoms. C2SR extends the algorithm in [Rubenstein et al., 2014] proposed for swarm robotic systems which assume different mechanical constraints. C2SR is a step toward realizing programmable matter. We evaluate our algorithm through simulation of large ensembles composed of more than ten thousand modules. We show the effectiveness of our algorithm and study its performance in terms of communications, movements and execution time. Our observations indicate that the number of communications, the number of movements and the execution time of our algorithm are highly predictable. Furthermore, we observe execution times that are linear in the size of the goal shape.

1.3/ OUTLINE

This dissertation is organized as follows. In Chapter 2, we present the context of this thesis, including the specific features of our target modular robotic systems. Research problems are then addressed in three separate chapters. In Chapter 3, we present our work on network centrality. Afterwards, we develop our work on time synchronization in Chapter 4. Our work on self-reconfiguration is presented in Chapter 5. These three contribution chapters are organized in a similar way. First, we briefly explain the research problem once more and then state the system model. Then, we provide a comprehensive overview of the state of the art on that problem. After that, we detail our contribution(s) and subsequently present experimental results before concluding the chapter. Finally, in Chapter 6, we summarize the contributions of this thesis and propose some directions for future work.

2

CONTEXT

Contents

2.1/ INTRODUCTION

In this chapter, we provide a contextual overview of the modular robotic systems and their applications. In this dissertation, we primarily target large-scale ones (LMRs), namely the large-scale lattice-based modular robots composed of resource-constrained identical modules that communicate together using only neighbor-to-neighbor communications. This chapter offers a network characterization of those systems along with a discussion of the challenges involved in the design of distributed algorithms for such systems. Finally, we present our research environment, i.e., the hardware and simulation tools we use to apply and evaluate our research.

2.2/ MODULAR ROBOTICS

In this section we introduce modular robotics. We first define this concept. Then, we present the advantages offered by modular robotic systems. Afterwards, we show some applications based on modular robotic systems. We then offer a classification of existing modular robotic systems. Finally, we discuss the network properties of LMRs.

2.2.1/ DEFINITION

Over the past decades, modular robotics has emerged as a new way to design robotic systems. A modular robot is formed from independent, intelligent and communicating modules which act as a whole ensemble. It forms a distributed system in which modules cooperatively self-organize, perform specific tasks and achieve common goals. Modular Self-Reconfigurable Robot (MSR) can rearrange their global shape to adapt to a task or a given situation.

2.2.2/ ADVANTAGES OVER TRADITIONAL ROBOTICS

Compared to traditional robotic systems, MSRs have four main advantages: versatility, robustness, extensibility and low cost. The versatility property directly comes from the fact that an MSR can self-adapt to a specific, possibly unexpected, situation by rearranging its global morphology. This enables modules to perform a wide variety of different tasks, including tasks not even envisaged at the time of designing. Modules are interchangeable both inside a robot and potentially with some surrounding systems. Hence, modular robotic systems are more robust, they may self-repair in case of module failure by discarding or replacing faulty modules on the fly. Moreover, modular robotic systems can be scaled up by adding/deleting modules when necessary. In addition, they also have economic advantages as a wide variety of different and complex systems can be built from mass-produced modules.

2.2.3/ EXAMPLES OF POTENTIAL APPLICATIONS

This section presents some interesting potential applications of modular robotic systems. To the best of our knowledge, none of the presented applications has been physically realized yet.

Conveyance System The Smart Blocks [Piranda et al., 2013] project aims to build a large distributed modular system to convey small and fragile objects, by attaching many modules together, each one equipped with a conveyance surface (see Figure 2.1). This surface can be deployed in inhospitable and remote locations (e.g., a remote planet, hazardous areas of a nuclear plant, etc.). The conveying system makes it possible to sort objects and transport them to different locations according to some criteria (e.g., shape, color, etc.). Moreover, if a module fails, the system can autonomously self-repair by replacing the faulty module by a functional one.

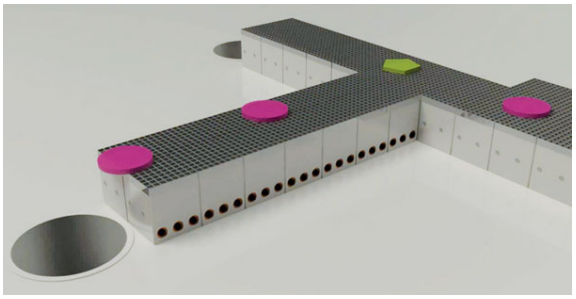


Figure 2.1: Smart conveyance surface formed from Smart Blocks. The system sorts the objects it distributively conveyes. Purple circles and green hexagons are transported toward two different holes.

Programmable Matter Programmable Matter (PM) is a matter that can change its physical properties in response to external and programmed events. Different approaches and technologies to realize PM are envisioned in the literature, e.g., PM using 4D printing [Tibbits et al., 2014], quantum wellstone [McCarthy, 2000], DNA structures [Ke et al., 2012, Kim et al., 2011] and robotic-based approaches. The latter include the use of self-folding robots [Hawkes et al., 2010], tendon-driven robotic chains [Lasagni et al., 2016], robotic materials [McEvoy et al., 2015], swarm robotic systems [Rubenstein et al., 2014] and modular self-reconfigurable robots [Goldstein et al., 2004, Gilpin et al., 2010].

In the Clay-Electronics (Claytronics) project [Goldstein et al., 2004], it is envisioned to use large-scale micro modular robotic systems, composed of up to millions of modules called Claytronics Atoms (Catoms), to build PM. Every Catom is a mass-producible micro robot that will have very restricted (i.e., strictly mandatory) functionalities. PM promises synthetic reality and has a wide range of applications (e.g., sending/downloading copies of physical objects, morpheable objects reshapable at will, injectable surgical instruments, 3D interactive life-size TV, etc.). It will enable people not only to control their environment but also to shape it.

As shown in Figure 2.2, PM offers, for instance, a drastic evolution of the computer-aided design process. In this vision, a computer holds a virtual representation of an object that can be transferred to some programmable matter in order to obtain a physical representation of that object. The virtual and the physical representations remain consistent at all times, i.e., if one changes, the other reflects this change. The user can modify the virtual representation and it will have an immediate impact on the physical representation of the object considered. He can also manually change the physical representation as he whishes, which will immediately update the virtual representation. Hence, designers will be able to simultaneously design a model and a prototype of their object, reducing significantly the time to prototype. Furthermore, the matter can be endlessly re-used and reshaped, thus this process will also minimize the waste of resources.

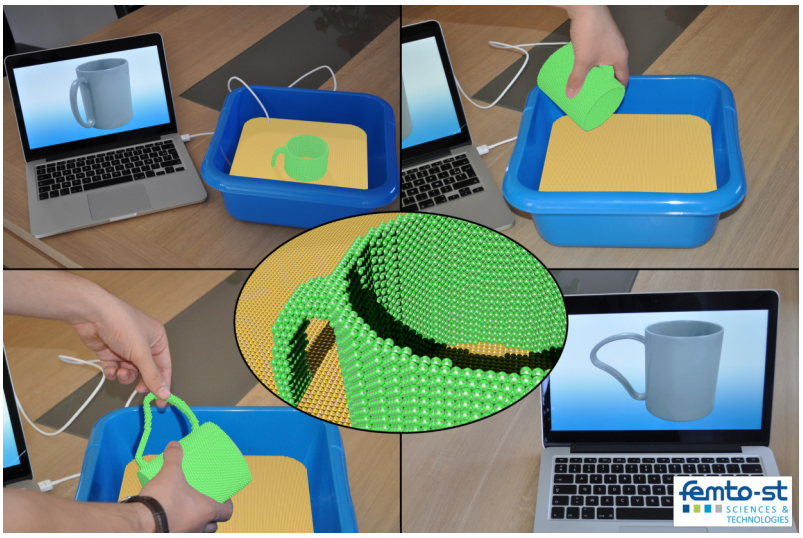


Figure 2.2: Programmable matter as a cyber-physical conjugation to enhance the computer-aided design process (from [Bourgeois et al., 2016]). The cyberized representation of a cup is transferred to the matter composed of hundreds of thousands of modules. The physical representation is then manipulated and manually modified. The cyberized representation remains consistent with the physical one and reflects the change.

Space Exploration Modular robotic systems can be used to overcome volume limitations in spacecraft during space exploration missions as explained in [Yim et al., 2009]. Modules can be packed in a dense way in order to meet vessel volume constraints and deploy at will during a mission to perform different tasks. Moreover, MSR-based objects can potentially self-repair, thus limiting the risk of a mission aborting in case of critical-equipment failure.

Search and Rescue Modular robotic systems may also be used in search and rescue operations in collapsed buildings, as explained in [Yim et al., 2009]. For instance, an MSR system can transform its shape to sneak in ruins and pass through narrow passages in

order to locate victims. Once a victim is found, the robot can emit a signal with its position and take the form of a shelter to protect the victim until rescued.

2.2.4/ Existing Systems and Classification

Existing modular robotic systems differ by their architecture (e.g., lattice, chain, mobile), their communication model (e.g., neighbor-to-neighbor communication, global communication through a shared medium, hybrid model), their module and overall scale (nanometer, micrometer, millimeter, centimeter, meter, etc.), their sensing and actuation (self-reconfigurable, manually reconfigurable, etc.) capabilities, etc. A comprehensive overview of the existing modular robotic systems can be found in surveys [Chennareddy et al., 2017, Ahmadzadeh et al., 2016, Yim et al., 2007]. The complexity that lies in the coordination of large-scale modular robotic systems depends on these hardware parameters [Yim et al., 2009].

In lattice-based modular robots, modules are arranged in some regular 2-dimensional or 3-dimensional lattice structures. In chain-based structures, modules are connected together in a serial manner forming an articulated chain or tree. By contrast, in mobile architectures, modules are free to move in the continuous space and can dock together to form lattice, chain or free structures.

In the neighbor-to-neighbor communication model, modules communicate only with adjacent modules. This communication model is fundamentally different than the global communication model where all modules can directly communicate together, for example, through a global bus. The later approach works well in small networks, but it is not scalable. Indeed, packet collisions may frequently occur. Moreover, if the shared communication medium is a bus, the number of hosts it can support is limited. Some hybrid approaches have been proposed but they are not common in modular robotics and complex to implement in a resource-constrained environment.

In this dissertation, we focus our attention on lattice-based modular robots composed of identical resource-constrained modules that communicate together using only neighbor-to-neighbor communications. We name this class of modular robotic systems LMRs. As shown in [Bourgeois et al., 2016], LMRs are particularly suitable to realize large-scale ensembles of modular robotic systems (e.g., Claytronics PM). Moreover, the class of modular robots considered captures a variety of existing systems, e.g., the Telecubes [Suh et al., 2002], the Miche [Gilpin et al., 2008] and the Distributed Flight Array [Oung et al., 2011] modular robots, some of the self-assembling systems used in [Bhalla et al., 2007] and most of the modular robotic systems developed in the Smart Blocks and the Claytronics projects. Figure 2.3 shows some LMRs developed in these two projects, namely the Smart Blocks, the millimeter-scale 2D Catoms, the Blinky Blocks and the 3D Catoms [Piranda et al., 2016b]. These modular robots are respectively arranged in the square, the hexagonal, the simple cubic, and the face-centered cubic lattices.

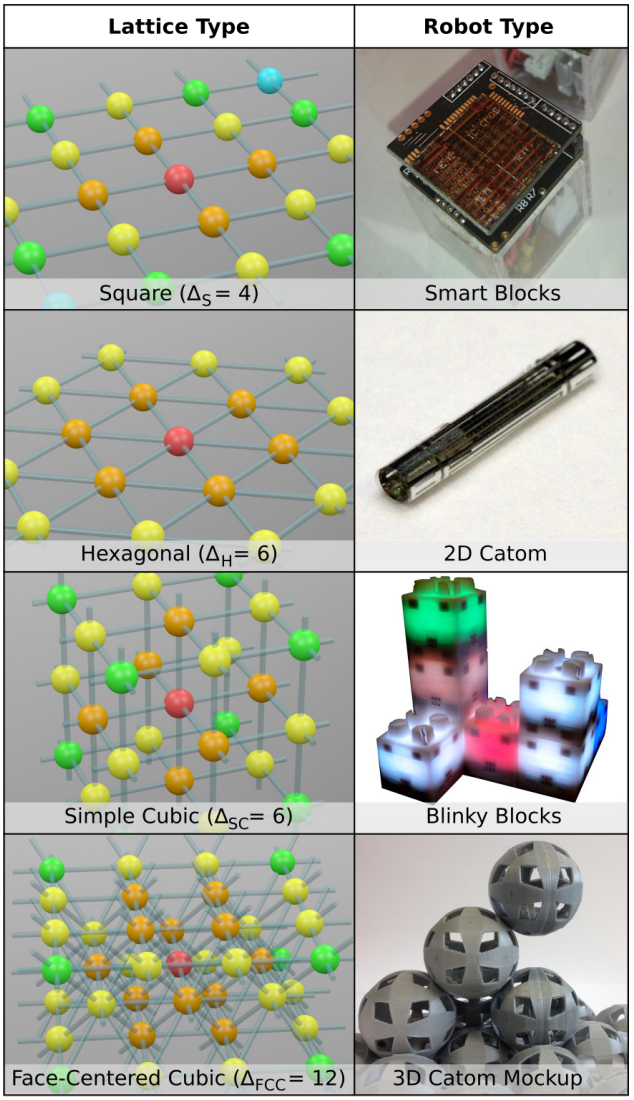


Figure 2.3: Different lattice arrangements associated with modular robotic systems developed in the Smart Blocks and the Claytronics projects. For a lattice L, Δ_L denotes its coordination number, i.e, the maximum number of modules to which a module can be connected.

2.2.5/ Network Properties of Large-Scale LMRs

In this section, we present key network characteristics of large-scale LMRs and discuss the challenges implied by these properties in the design of efficient distributed algorithms for large-scale ensembles.

- **Restricted Resources:** Nodes are low-cost small electronic devices. Thus, they are equipped with limited capabilities. They have scarce computation, memory and energy resources. They may also have, for instance, low-precision clocks making distributed real-time control a difficult task.
- **Asynchrony:** Modules of LMRs are inherently asynchronous. Indeed, there is no global clock and every module processes independently of the others. In particular,

communication between modules is asynchronous.

- **Neighbor-to-Neighbor Communications:** In the neighbor-to-neighbor communication model, a module uses a separate network interface and communication channel for each of its neighbors. The network has neither local nor global shared broadcast medium. A remarkable advantage of the absence of shared communication medium is that we do not have to deal with potential network collisions. In our model, a module can communicate simultaneously with all its neighbors. Moreover, in order to locally broadcast a message, a module has to send an individual copy of that message to all neighbors. Although trivial, these two properties have to be taken into account when designing algorithms at risk of overwhelming the network. For instance, if all nodes simultaneously start a network flooding operation, a node may generate messages at a higher rate than it can send them. A node may receive a message per neighboring node in a short amount of time, thus adding several messages in the channel to each of its other neighbors. A short amount of time later, the same node might again receive a message from all its neighbors and add messages in its outgoing message queues, although only a part of the messages previously inserted into the outgoing queue has been sent. Thus, messages progressively pile up. If this situation occurs several times, the outgoing queues keep growing and the network gets congested. This issue is further discussed in Section 3.5.1.2.
- **Sparse Networks:** We demonstrate in Appendix A that LMRs form sparse networks, i.e., $m \ll n^2$, where n is the number of modules in the system and m the number of links in the network. Moreover, we show that the number of links is $\Theta(n)$. We compare lattice-based networks to small-world networks [Watts et al., 1998] (e.g., the Internet network [Jin et al., 2006]) and to wireless ad-hoc networks (e.g., wireless sensor networks, multi-robot networks, etc.). Since many large real-world networks are small-world networks, it is legitimate to consider them for comparison. Wireless ad-hoc networks are highly spatially dependent, like our class of networks. Indeed, in wireless ad-hoc networks, nodes can only communicate with some neighboring nodes within some limited range. Note that wireless ad-hoc networks can fall in the class of lattice-based networks if they are deployed in a lattice structure. Lattice-based networks are sparser than small-world networks that have $\Omega(n\log(n))$ edges [Watts et al., 1998]. Wireless ad-hoc networks can be sparse or dense, depending on the deployment environment (area/volume, obstacles, etc.), the deployment density and the node communication range. An example of sparse sensor network is the 46-hop network of 64 sensors deployed in a long-linear topology on the Golden Gate Bridge, in San Francisco (United States), in order to monitor the effects of wind and earthquakes on the structure [Kim et al., 2007].
- **Large Hop Distances:** In systems where nodes use neighbor-to-neighbor communications, the node spatial arrangement directly reflects the connectivity graph. Modular robotic systems often have a bounded number of connectors, i.e., of potential neighbors. As a direct consequence, large-scale modular robotic ensembles tend to exhibit large hop distances. Due to the regular tiling of the space in lattices, networks of LMRs obey certain geometric rules. In regular lattice networks, the typical distance between two nodes is $\sim n^{\frac{1}{D_L}}$ [Barthélemy, 2011] where

D_L is the geometric dimension of the lattice L. Thus, in lattice-based networks, i.e., lattice networks with potential holes, this distance is lower bounded by $\Omega(n^{\frac{1}{D_L}})$, while in small-world networks, this distance is $\sim log(n)$ [Barthélemy, 2011]. In Appendix A, we provide exact bounds on the radius and the diameter of these networks based on their lattice type and the number of modules in the system. Moreover, we demonstrate that the radius and the diameter of lattice-based networks are lower bounded by $\Omega(\sqrt[3]{n})$. Small-world networks have typically short distances between arbitrary pairs of nodes due to the presence of a few long-range edges. As a consequence, small-world networks tend to have a small diameter. In lattice-based and sparse wireless ad-hoc networks, such long-range edges do not exist. Thus, these networks tend to have a larger average distance and a larger diameter. These phenomena are accentuated as the number of nodes in the network increases. Studies indicate that the diameter of the Internet is around 30 hops [Latapy et al., 2006, Leguay et al., 2005, Cardozo et al., 2012]. This is corroborated by the suggested values for Time-To-Live (TTL) for Internet Protocol (IP) packets. The TTL should be twice the diameter of the Internet [Braden, 1989] and the actual value recommended is 64 [Reynolds et al., 1994, The Internet Assigned Numbers Authority (IANA), 2016]. As shown in Figure 2.4, systems with a million 3D Catoms have a diameter of at least 132 hops, while systems with 100 million 3D Catoms have a diameter of at least 620 hops. Similarly, Blinky Blocks systems have a large diameter, e.g., a 40,000 Blinky Blocks system has a diameter greater than 30 hops. Thus, a 40,000 Blinky Blocks system which fits in a $1.4\ m^3$ cube, would have a diameter larger than the entire Internet that spans the whole world. It is crucial to take into account the large diameter and large average distance to design efficient and effective distributed algorithms for large-scale modular robotic systems. For example, communication over a large number of hops causes latency and reliability issues. Let us consider time synchronization and data sharing algorithms. These algorithms are, for instance, required for real-time responsive programmable matter and to distribute, store and access geometry data for self-reconfiguration. However, these algorithms are challenging to design for such large-diameter and large-average-distance systems. Unpredictable delays (due, for example, to queueing or retransmissions) accumulate every hop, which tends to disturb the time synchronization process and decrease the achievable synchronization precision. Moreover, in data sharing algorithms, lookup latency may be extremely long if it involves messages that have to travel a large number of hops.

2.3/ Research Environment: Evaluation Hardware and Simulation Tools

This section presents the hardware and simulation tools we use to apply and evaluate the contributions introduced in this thesis. We consider the Blinky Blocks [Kirby et al., 2011] and the 2D Catoms [Karagozler et al., 2009, Karagozler, 2012] modular robotic systems. We first present technical features of these two systems.

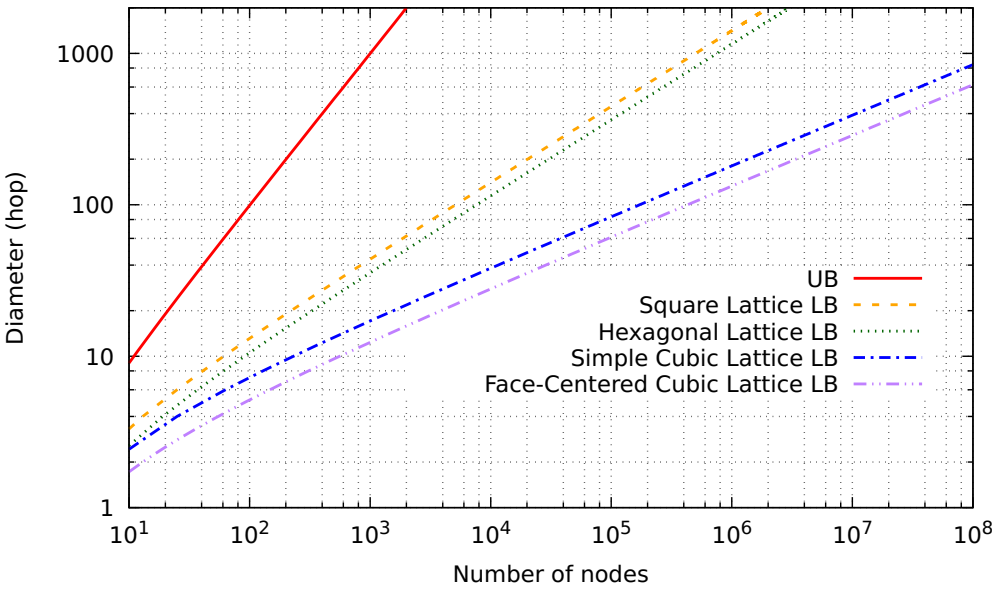


Figure 2.4: Diameter bounds versus the number of nodes in the network for the different lattices considered. The terms “LB” and “UB” respectively stand for “lower bound” and “upper bound”.

In the next chapters, we evaluate our algorithms using both hardware prototypes and simulations. We have at our disposal several dozens of hardware Blinky Blocks to perform experimental evaluations. In order to carry out evaluations on 2D Catoms systems and on large-scale Blinky Blocks systems, we use VisibleSim [Dhoutaut et al., 2013], our simulator of modular robotic systems. This section also presents VisibleSim.

2.3.1/ BLINKY BLOCKS

Blinky Blocks are centimeter-size blocks that were developed in the Claytronics project. Figure 2.5 shows the details of a single block and an example of program running on an ensemble of hardware Blinky Blocks. We have at our disposal a few dozen Blinky Blocks to evaluate our algorithms on real hardware.

Blocks are attached to each other using magnets. Each module has its own computational power provided by an ATMEL ATxmega256A3-AU 8/16-bits 32-MHz microcontroller having 256KB ROM and 16KB RAM [ATMEL, 2013], as well as sensors and actuators such as RGB leds to glow with different colors according to the programmer’s will.

All the blocks of a system execute the same program. A single block is connected to a power supply. Power is distributed through the system using dedicated pins. A block can have up to 6 neighbors and can communicate with them through serial links on the block faces. Ensembles of Blinky Blocks are manually reconfigurable at will. Blinky Blocks use full-duplex neighbor-to-neighbor communications over serial links controlled by Universal Asynchronous Receivers/Transmitters (UARTs) configured with a bitrate of 38.4 kBauds.

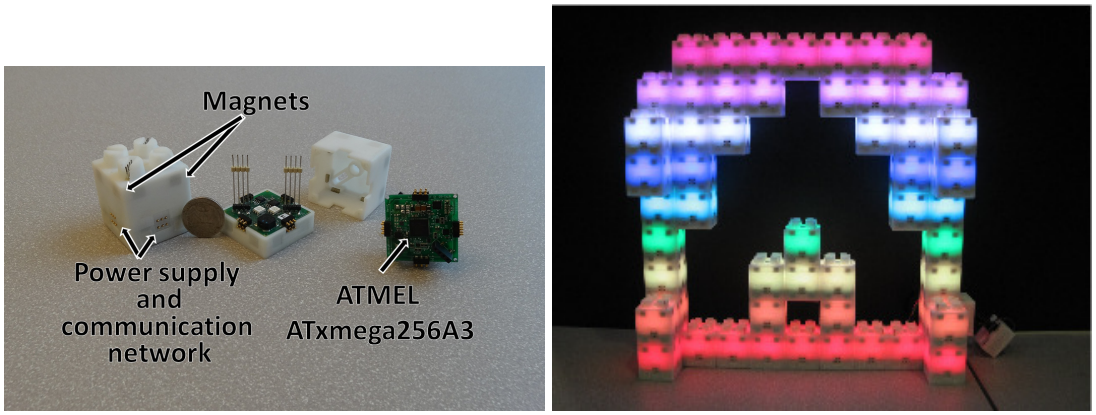


Figure 2.5: On the left, dissection of a Blinky Blocks hardware prototype. On the right, an ensemble of 58 Blinky Blocks hardware prototypes running the Rainbow program (from [Kirby et al., 2011]). In the Rainbow program, blocks are colored depending on their level in the structure.

Modules exchange frames that contain up to 17 bytes of data. Furthermore, a distributed logging system enables all modules to send information to a computer connected to the system using a serial connection.

More details on the Blinky Blocks communication system along with the characteristics of the block hardware clocks are provided in Section 4.6.

2.3.2/ 2D CATOMS

2D Catoms are millimeter-scale cylindrical robots [Karagozler et al., 2009, Karagozler, 2012] developed in the Claytronics project. 2D Catoms have been partially validated with the realization of a hardware prototype (see Figure 2.6).

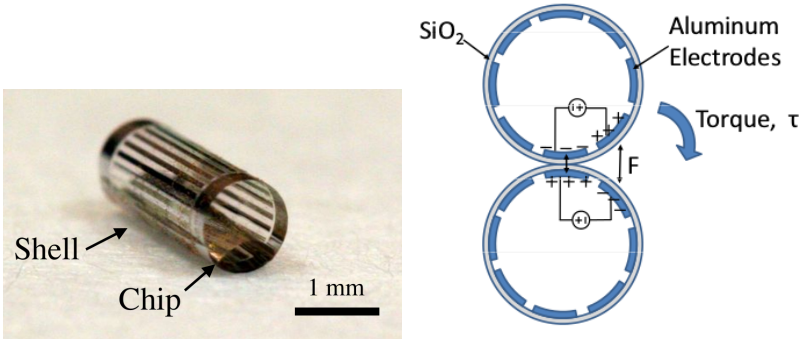


Figure 2.6: 2D-Catom prototype (from [Karagozler, 2012]).

A 2D Catom consists of a 6-mm long-and 1-mm-diameter cylindrical shell. A high-voltage CMOS die is attached inside the tube. The chip includes a storage capacitor and a simple logic unit. The tube has electrodes used for power transfer, communications and actuation.

In our work, we assume the power is spread from a powered floor through the ensemble using neighbor-to-neighbor power transfer. We consider that 2D Catoms are organized into a horizontal pointy-topped hexagonal lattice where modules have up to six neighbors. Modules can communicate together using neighbor-to-neighbor communications.

Moreover, a 2D Catom can roll Clockwise (CW) or Counter-Clockwise (CCW) around a stationary module. During an atomic move, a module rotates 60, going from one cell of the lattice to its adjacent cell. We assume that a 2D Catom has only the capability to lift itself, it cannot carry or push other modules. In the current design, a 2D Catom is theoretically able to perform a revolution in 1.67 seconds or 3.35 seconds [Karagozler, 2012], which corresponds to an average speed[1] of $1.88\ mm \cdot s^{-1}$ or $0.94\ mm \cdot s^{-1}$.

2.3.3/ VISIBLESIM

The VisibleSim simulator [Dhoutaut et al., 2013] is a discrete-event simulator for modular robots developed in our team (see Figures 2.7 and 2.8).

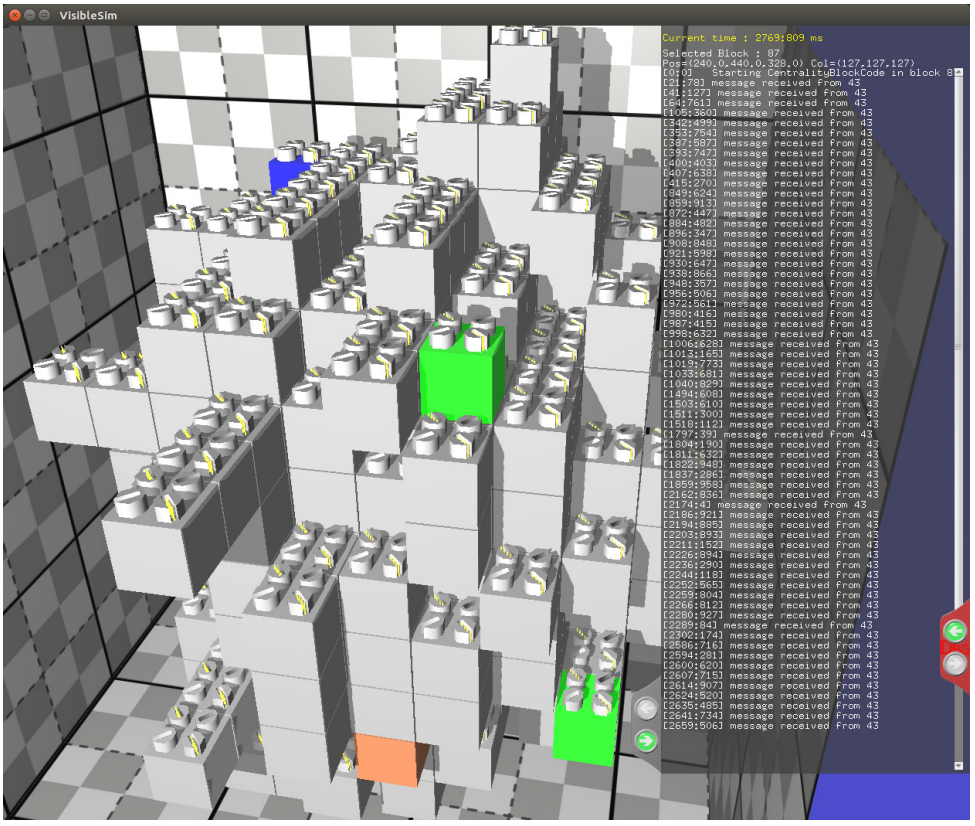


Figure 2.7: Screenshot of VisibleSim simulating the execution of the ABC-CenterV1 algorithm (see Section 3.7) in an ensemble of 500 Blinky Blocks.

[1] Let t_r and t_u respectively denote the time for a complete revolution and the time for a unit movement. In a revolution, a d-millimeter diameter cylindrical catom horizontally travels πd millimeters. Hence, $v = \frac{d\pi}{t_r}$. For $t_r = 1.67s$, $v = \frac{d\pi}{t_r} = 1.88mm \cdot s^{-1}$. In a unit movement, this catom travels $\frac{1}{6} \times \pi d = 0.523mm$. Thus, $t_u = \frac{0.523}{v}$. Note that t_u can be computed without determining v. Indeed, $t_u = t_r \times \frac{60}{360}$. For $t_r = 1.67s$, $t_u = 0.278s$.

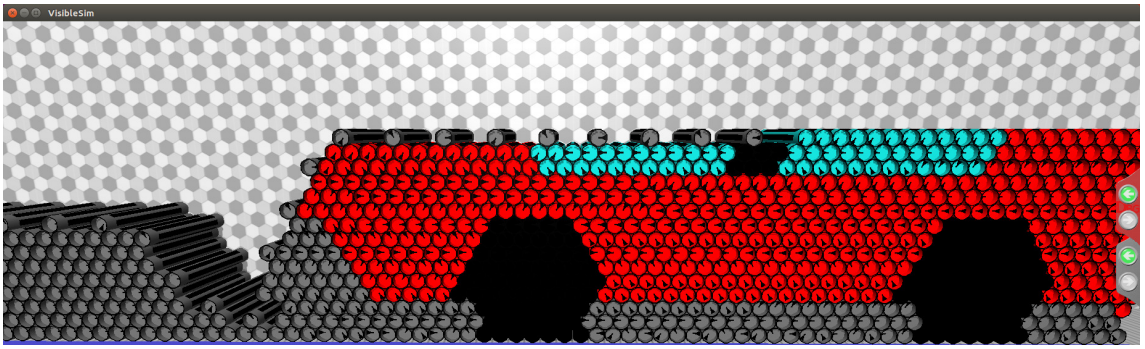


Figure 2.8: Screenshot of VisibleSim simulating the execution of the Cylindrical Catoms Self-Reconfiguration (C2SR) algorithm in an ensemble of 1073 2D-Catoms (see Section 5.4).

VisibleSim supports a variety of different modular robotic systems (e.g., the Blinky Blocks, the 2D Catoms, the Smart Blocks, the 3D Catoms). We use VisibleSim to simulate the behavior of algorithms on modular robotic ensembles and also to benchmark their performance in terms of execution time, communications, number of motions, etc.

VisibleSim enables to perform experiments on large-scale ensembles as it can handle simulations with dozens of thousands of modules. VisibleSim also allows us to carry out experiments on modular robotic systems for which we do not have fully functional hardware prototypes at our disposal (e.g., the 2D Catoms).

To properly simulate system asynchrony, VisibleSim can be run with variable motion and communication models. The simulation models used in the evaluation of the contributions of this thesis are chapter-specific and thus detailed later on in the evaluation section of the different contribution chapters.

2.4/ CONCLUSION

In this chapter, we provide a short overview of modular robotics. In addition, we present the hardware and simulation tools we use to apply and evaluate the contributions introduced in this thesis.

Moreover, we show that large-scale LMRs form asynchronous, low-degree, sparse, large-average-distance and large-diameter networks. In addition, units have limited computation, memory and energy resources. It is important to take into account these properties to design efficient and effective distributed algorithms for large-scale ensembles. In the next chapter, we propose algorithms to distributively elect a central module that is well located to communicate with all the others.

3

Centrality-Based Leader Election

Contents

3.1/ INTRODUCTION

In this chapter, we present our work on network centrality. Distributed systems are composed of independent connected nodes that coordinate their activities through communications in order to achieve common goals. Coordination in distributed systems often requires a single node to act as a leader and to perform some specific roles in the system. We address the issues of effectively and efficiently electing an approximate-centroid node or an approximate-center node in distributed embedded systems.

The centroid is the set of nodes of minimum average distance to the others while the center is the set of nodes of minimum maximum distance to the others. These sets of nodes exhibit interesting properties for distributed system applications. For instance, centroid nodes are ideal nodes for hosting query-oriented service providers. Indeed, assuming that queries are likely to originate from any node in the network, placing service providers at the centroid minimizes the expected traveling distance for queries and answers, which implies low average time delays and message costs. To elect such nodes in an arbitrary asynchronous network, classical distributed algorithms require complete information about the network topology. Therefore, these algorithms are not scalable and not suitable for distributed embedded systems with limited computational, memory and energy resources.

Since modular robots form shapes, a first intuition is to use a geometric approach by computing the centroid of the configuration. In Figure 3.1, the *geometric centroid* is the point C that stands in the middle of the object. Blocks in red represent the set of blocks that minimizes the worst-case network distance to all the other blocks. As we can see, the geometric centroid corresponds to a block in red for the grid shape and the S-line shape. However, for the torus, there is no block present at the location of the geometric centroid and all the blocks have the same worst-case distance. The case of the G-shape is even worse, as the geometric centroid of the shape leads to the worst case in terms of network distance. The S-line and G-line shapes are similar in the sense that they form a topological line and the block that minimizes the worst-case network distance stands in the middle of this line. There is an obvious mismatch between geometrical distances and network ones. Therefore, we need to consider network topologies instead of geometrical shapes and to use computations based on network distances instead of geometric ones.

Note that geometrical information could potentially still be used as a hint to start central node computation. Nevertheless, we decided to not use such information, in order to design generic distributed algorithms that do not rely on geometry.

In this chapter, we consider a rather general system model. We assume a distributed system formed from an asynchronous non-anonymous point-to-point unweighted and undirected network in which nodes can only communicate with their immediate neighbors (neighbor-to-neighbor communication model). The complete system model is defined in Section 3.2.

The contribution of this chapter is to propose a collection of both efficient and effective distributed algorithms to elect approximate-centroid and approximate-center nodes in asynchronous distributed systems. We propose the ABC-Center algorithm, the k-BFS Sum-

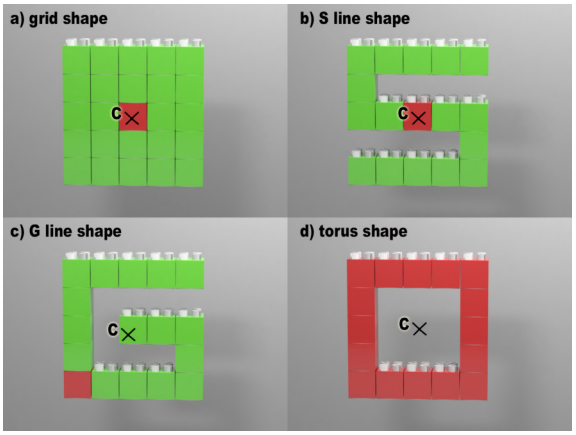


Figure 3.1: Difference between the geometric centroid (represented by C) and the Jordan center (in red). The Jordan center is defined as the set of nodes of minimum maximum distance to the others (see Section 3.3).

Sweep framework and the Probabilistic-Counter-based Central-Leader Election (PC2LE) framework. Frameworks are declined in two versions, one for approximate-center node election, another for approximate-centroid node election. Our algorithms and frameworks do not require any prior knowledge of the network, have a well-defined termination criterion, converge in a reasonable amount of time and are memory-efficient. The source code of our algorithms is available online[1,2].

In the k-BFS SumSweep, nodes compute their partial centrality value to a subset of root nodes composed of a random initial node and $k-1$ most external nodes. Root nodes are consecutively selected using the SumSweep approach that was originally proposed in the sequential algorithm for the exact radius and diameter computation of external graphs [Borassi et al., 2014]. Distributed Breadth-First Searches (BFSes) are used for distributed Single-Source Shortest Paths (SSSP) computations. The main idea behind our framework is that central nodes are first and foremost central to the most external nodes.

ABC-Center extends the sequential Minimax [Handler, 1973] and 4-Sweep [Crescenzi et al., 2013] algorithms[3]. ABC-Center identifies an extreme path and recursively isolates midpoints on it until electing a single module. The main idea of ABC-Center is that central nodes lie in the middle of a diameter path. ABC-Center may be more convenient to use than the k-BFS framework as ABC-Center converges by itself, i.e., its termination does not rely on any input parameter.

PC2LE is based on the input-graph analysis algorithms [Kang et al., 2011a, Kang et al., 2011b] and the distributed synchronous algorithm [Garin et al., 2012] which use low-

[1]GitHub repository that hosts our algorithm codes for simulations: https://github.com/nazandre/thesis

[2]Official Blinky Blocks firmware repository in which some of our algorithm codes are hosted: https://github.com/claytronics/oldbb. At the time of submitting the final version of this manuscript, the k-BFS SumSweep and ABC-CenterV2 algorithms have been implemented for hardware Blinky Blocks.

[3]Some examples of ABC-Center executions on Blinky Blocks systems are available online in video at https://youtu.be/QxK12UAq42o and https://youtu.be/PYnJn6tXKa8

memory-footprint probabilistic counters (e.g., Flajolet-Martin [Flajolet et al., 1985], HyperLogLog [Flajolet et al., 2007]) to estimate node centrality measures. In PC2LE, an estimated centrality value is computed for all nodes. PC2LE is approximately equivalent to running a BFS from every node, but at less expense in terms of computations and communications.

To test our algorithms and frameworks, we apply them to the Blinky Blocks (see Section 2.3.1). Although we use modular robots to present and evaluate our algorithms, they work on all distributed systems that satisfy the assumptions detailed in Section 3.2. We evaluate our algorithms both on hardware prototypes and through simulations. Experimental results show that our algorithms scale well accuracy, execution time, number of messages and memory usage. To the best of our knowledge, our algorithms are the most precise distributed algorithms to elect an approximate centroid or an approximate center in our target systems, with both a reasonable convergence time and a limited storage cost.

This chapter is organized as follows. In Section 3.2, we define the system model. Afterwards, we provide a comprehensive overview of the existing centrality measures and definitions. Then, we discuss the related work in Section 3.4. In Sections 3.6, 3.7 and 3.8 we respectively detail the k-BFS SumSweep framework, the ABC-Center algorithm, and the PC2LE framework. In Section 3.9, we provide experimental results. In Section 3.11, we conclude this chapter.

3.2/ System Model and Assumptions

System Model In this chapter, we consider distributed systems forming asynchronous non-anonymous point-to-point unweighted and undirected networks in which nodes can only communicate with their immediate neighbors (neighbor-to-neighbor communication model). Every node v_i has a unique identifier, id_{v_i}. We assume that communication channels are FIFO (first in first out) and bidirectional , i.e., messages are received in the order in which they have been sent and the channels can carry messages in both directions (as in Section 1.1.1 of [Raynal, 2013]). Similarly to [Awerbuch, 1985], we further consider that messages have a bounded length and may carry only a limited amount of information. Each message sent by a node to its neighbor arrives within some finite but unpredictable time.

Distributed algorithms often involve a resource-performance trade-off (e.g., memory usage, execution time, communication). We make design choices considering that our algorithms target large-scale Distributed modular robotic ensemble composed of resource-constrained identical modules that are organized in a lattice structure and communicate together using only neighbor-to-neighbor communications. (LMR) ensembles.

Notation These systems can be modeled by an undirected and unweighted graph of inter-connected entities $G = (V, E)$, where V is the set of vertices (representing the nodes), E the set of edges (representing the connections), $|V| = n$, the number of vertices, $|E| = m$,

the number of edges. $d(v_i, v_j)$ refers to the distance between vertices v_i and v_j, i.e., the number of edges on a shortest path between v_i and v_j. The diameter, d, of the network is defined as $d = \max_{v_i \in V} \max_{v_j \in V} d(v_i, v_j)$. Δ is the maximum network degree, i.e., the maximum number of neighbors that a node has in the network.

$N_{v_i}(h)$ represents the set of nodes within distance h hops from node v_i and $N^h_{v_i}$ represents the set of nodes exactly at distance h hops from v_i. We assume that every node v_i has a unique identifier id_{v_i} and maintains a consistent list of its immediate neighbors $N^1_{v_i}$ using an external link-layer protocol. A message loss is considered to be a link failure and thus a neighbor departure.

Note on Complexity Calculation Unless otherwise mentioned, memory complexities are expressed in terms of machine words rather than in bits. Hence, we consider that a variable of a primitive data type (integer, boolean, etc.) uses $O(1)$ memory space. The number of values that can be encoded using variables may, however, induce limitations on the system size. For example, if node identifiers are encoded on w bits, the system may contain at most 2^w nodes.

The memory usage of a distributed algorithm is composed of both its application layer memory requirements and the space it needs to store messages. For instance, if during the execution of an algorithm that uses $O(1)$ space at the application layer, a module may simultaneously receive or send up to one message from/to all neighbors, this algorithm has a memory complexity of $O(\Delta)$.

Unless explicitly mentioned, we take into account message pileups in time and memory calculations.

3.3/ Network Centrality Metrics and Definitions

Graph and network centrality have been extensively studied in various domains such as in biology to identify the oldest metabolites [Wuchty et al., 2003], in social networks to find the most influential persons [Hanneman et al., 2005], in computer networks to elect the most suitable root node to start message broadcasting [Korach et al., 1984], etc. Many metrics and definitions of centrality have been proposed. This section offers an overview of the most commonly used and their possible applications.

3.3.1/ Definitions

The *Jordan center* [Wasserman, 1994] is the set of all nodes of minimum eccentricity, where the eccentricity $ecc(v_i)$ of a node v_i is the maximum distance from v_i to any other node (see Equations (3.1),(3.2) and (3.3)). The inverse of the eccentricity is sometimes called the graph centrality [Lehmann et al., 2003]. In this work, we use the term *center* to refer to the *Jordan center*.

$$ecc(v_i) = \max_{v_j \in V} d(v_i, v_j) \tag{3.1}$$

$$= \min \operatorname*{argmax}_{r \in \mathbb{N}} |N_{v_i}(r)| \tag{3.2}$$

$$Jordan\ Center = \{v_c \in V \mid ecc(v_c) = \min_{v_i \in V} ecc(v_i)\} \tag{3.3}$$

The *centroid* [Dutot et al., 2011] is the set of all nodes of minimum farness, where the farness $far(v_i)$ of a node v_i is the sum of the distances to all the other nodes (see Equations (3.4) and (3.8)). The farness can be equivalently computed using the size of the sets of nodes at increasing hop distances [Kang et al., 2011a] (see Equation (3.5) and (3.6)).

The centroid can be equivalently defined as the set of all nodes of maximum closeness, where the closeness $clo(v_i)$ [Freeman et al., 1979] of a node v_i is the inverse of its farness (see Equations (3.7) and (3.9)). The centroid can also be seen as the set of all nodes of minimum average distance to the others. The centroid is also called the *barycenter* [Mamei et al., 2005] or the *median* [Korach et al., 1984] of the graph.

$$far(v_i) = \sum_{v_j \in V} d(v_i, v_j) \tag{3.4}$$

$$= \sum_{r=1}^{d} r \times |N^r_{v_i}| \tag{3.5}$$

$$= \sum_{r=1}^{d} r \times (|N_{v_i}(r)| \ - \ |N_{v_i}(r-1)|) \tag{3.6}$$

$$clo(v_i) = \frac{1}{far(v_i)} \tag{3.7}$$

$$Centroid = \{v_c \in V \mid far(v_c) = \min_{v_i \in V} \ far(v_i)\} \tag{3.8}$$

$$= \{v_c \in V \mid clo(v_c) = \max_{v_i \in V} \ clo(v_i)\} \tag{3.9}$$

$$\tag{3.10}$$

The *center of gravity* or the *center of mass* [Dutot et al., 2011] is the set of all nodes of minimum weight, where the weight $we(v_i)$ of a node v_i is the average of the squared distances to all the other nodes (see Equations (3.11) and (3.12)). In a graph with positive distances, the center of mass is equivalent to the centroid.

$$we(v_i) = \frac{1}{n} \sum_{v_j \in V} d(v_i, v_j)^2 \tag{3.11}$$

$$Center\ of\ mass = \{v_c \in V \mid we(v_c) = \min_{v_i \in V}\ we(v_i)\} \tag{3.12}$$

The degree centrality metric [Freeman et al., 1979] is based on the number of links a node possesses (see Equation (3.13)). It is straightforward to compute it, but it only captures local information. This metric is not relevant in modular robots where modules have a bounded number of neighbors. Indeed, many of the modules will usually have the maximum number of neighbors a module can have.

$$deg(v_i) = \mid \{e \mid e = (v_i, v_j) \in E, v_j \in V\} \mid \tag{3.13}$$

The betweenness centrality metric [Freeman et al., 1979] is based on how much a given node belongs to the shortest path of other nodes (see Equation (3.14)). $\sigma_{v_j v_k}$ is the total number of shortest paths from node v_j to node v_k and $\sigma_{v_j v_k}(v_i)$ represents the number of those paths that pass through v_i. We choose to name the set of nodes of minimum betweenness the *betweenness center* (see Equation (3.15)).

$$bet(v_i) = \sum_{\substack{v_j, v_k \in V \\ v_i \neq v_j \neq v_k}} \frac{\sigma_{v_j v_k}(v_i)}{\sigma_{v_j v_k}} \tag{3.14}$$

$$Betweenness\ center = \{v_c \in V \mid bet(v_c) = \max_{v_i \in V}\ bet(v_i)\} \tag{3.15}$$

For the sake of brevity, other centrality measures proposed in the literature (e.g., the stress centrality [Shimbel, 1953] that reflects the volume of traffic that passes through a given node, the eigenvector centrality [Bonacich, 1972] which measures the influence of a node in a network, etc.) fall beyond the scope of this chapter. Recently, some low-complexity centrality measures that aim at approximate common centrality measures have been proposed (e.g., the tree-based centrality [Kim et al., 2013], the localized bridging centrality [Nanda et al., 2008], etc.). We choose to consider them as approximation algorithms and present those related to our work in the next section.

3.3.2/ Properties and Applications

Figure 3.2 illustrates the differences between the different notions of center. The Jordan center is strongly influenced by the diameter of the system. The betweenness center is sensitive to critical paths between large sets of modules. Indeed, the modules that connect the two large squares are on all the paths between any module of the two squares.

In the context of distributed system applications, each type of central node has its own interesting properties. We assume that messages travel along the shortest paths. The Jordan center is suitable as initiator of parallel communications to all the other nodes, e.g.,

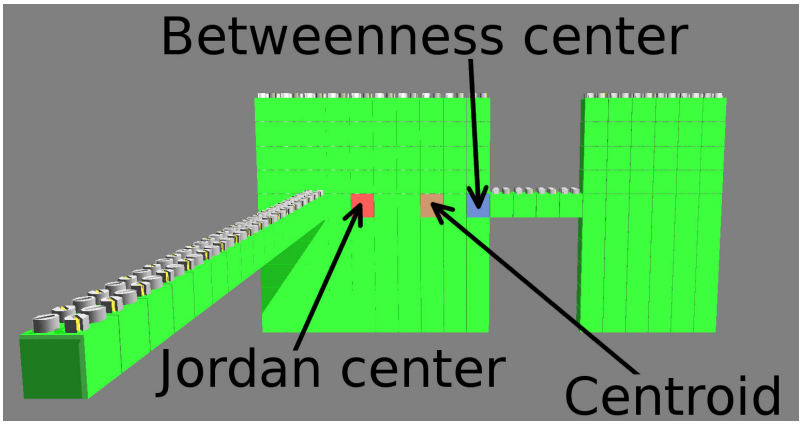


Figure 3.2: Differences between the different types of central module in a Blinky Blocks system.

network flooding of broadcast messages. Flooding from the Jordan center minimizes the maximum traveling distance of the messages, which implies low maximum time costs. Centroid is ideal for unicast communications with all the other nodes, e.g., query-oriented services. Assuming that queries are likely to originate from any node, placing service providers at the centroid of the network minimizes the expected traveling distance for queries and answers, which implies low average time and message costs. The betweenness center is most useful for controlling and analyzing the network traffic. Indeed, the betweenness centrality of a node reflects the proportion of traffic that passes through this node. As a consequence, this measure favors nodes that join communities (i.e., dense subnetworks), rather than nodes that lie inside a community. The betweenness centrality can also be interpreted as a congestion sensitivity measure [Lehmann et al., 2003].

Time master position	Maximum pairwise difference (ms)	Mean absolute difference (ms)
extremity (front of the left arm)	49.00	2.78
center	28.00	2.39
centroid	33.00	2.37
betweenness center	35.00	2.42

Table 3.1: Impact of the position of the time master on the synchronization error in an enlarged version of the system depicted in Figure 3.2. The system is synchronized using the Modular Robot Time Protocol (see Section 4.5). This system is composed of 1,456 nodes and has an 83-hop diameter. Every module in the system of Figure 3.2 is actually enlarged in a cube of 2x2x2 modules in this experiment. Results were computed on 3.5-hour-long simulations during which the synchronization error was measured every 3 seconds.

Table 3.1 shows the impact of the position of the time master on the synchronization error in an enlarged version of the Blinky Blocks system depicted in Figure 3.2. The

system is synchronized using the Modular Robot Time Protocol (see Section 4.5). As shown, placing the time master at a central node definitely leads to more synchronization precision. Moreover, placing the time master at the center (resp. centroid) tends to minimize the maximum (resp. average) synchronization error.

As shown in this section, existing types of central nodes have different features and applications. In this chapter, we propose efficient and effective algorithms to elect an approximate-center node or an approximate-centroid node.

3.4/ State of the Art

As explained in the previous section, several types of centrality definitions exist. For conciseness reasons, we restrict our study to the work related to the center and to the centroid. For criticality centrality measures, the reader can refer to [Nanda et al., 2008, Tizghadam et al., 2010, Kermarrec et al., 2011, Kang et al., 2011a].

Existing algorithms for centrality computation can be categorized into four major families, namely exhaustive, graph-specific, sampling-based and probabilistic-counter-based. Other proposed approaches include tree-based computations, random-walk-based methods and linear programming approaches.

Centrality computation is an active research topic in both the graph analysis and the distributed system communities. They address computation on graphs with two different perspectives, namely input-graph analysis and distributed computation on the network graph. In input-graph analysis algorithms, one or several computers perform calculations on (external) graphs provided as input. These algorithms can be sequential or, for higher performance, parallel and/or distributed. In distributed graph algorithms, the graph is the network itself and the nodes cooperatively self-perform computation on it, in a distributed fashion. These algorithms generally do not require nodes to hold a global view of the networks. This work is about distributed graph algorithms.

Only the algorithms for asynchronous distributed systems match our system model, but we still present the different approaches as they are closely related to our problem. We consider that recent advances in graph analysis should be taken into consideration to design efficient and scalable distributed algorithms.

3.4.1/ Exhaustive Methods

Exhaustive methods are exact and involve a distributed All-Pair Shortest Paths (APSP) computation. We first discuss the APSP problem and then present exhaustive approaches to compute node centrality.

Different methods exist to solve the APSP problem in asynchronous networks. APSP can be computed using the distributed Floyd–Warshall's shortest path algorithm [Toueg, 1980] which runs in $O(n^2)$ time using $O(n^3)$ messages with $O(n)$ messages that carry $O(n)$ distances [Raynal, 2013]. APSP can also be computed using BFSes. Performing a sin-

gle BFS using Cheung's algorithm [Cheung, 1983] takes $O(d)$ time, if we ignore message pileups, and uses $O(nm)$ messages [Raynal, 2013]. All nodes can initiate a BFS traversal in parallel. However, the network may get congested, since messages will pileup, thus incurring a large time and memory overhead. On the other hand, BFSes can be performed one by one but it is expensive in terms of time. It uses in total $O(nd)$ time and $O(\Delta)$ space per node if message pileups are ignored. Also note that computing all the distances in parallel require the storage of $O(n)$ distances per node while, in sequential approaches, only the distance to the current-BFS root along with the partial farness/eccentricity are stored per node and progressively updated.

An almost asymptotically optimal distributed synchronous APSP algorithm has been proposed in [Holzer et al., 2012]. In this algorithm, a node triggers a BFS traversal one time unit after having been visited by a depth-first search traversal. This ensures that BFSes do not collide. Thus, to compute its eccentricity/farness, a node only needs to store information about a single BFS at a time. This algorithm runs in $O(n)$ synchronous rounds.

In [Korach et al., 1984], the authors propose algorithms to distributively elect the center and the centroid of graphs in different settings (asynchronous and synchronous networks, tree and arbitrary networks). The algorithms for asynchronous arbitrary networks use an exhaustive approach in which an initiator orders all nodes, through a depth-first search traversal of a spanning-tree, to compute their centrality value (eccentricity or farness), and then elects a node of minimum centrality value over the spanning tree. Any shortest path algorithm can be used to compute the distances from one node to all the others.

A distributed algorithm designed to elect the network centroid using n parallel breadth-first network traversals without acknowledgment was proposed in [Mamei et al., 2005]. Each node initiates a BFS by broadcasting a message that contains a hop counter increased at each hop. The authors claim that since the farness decreases monotonically to the centroid, nodes determine locally whether they are in the centroid or not by comparing their farness value to those of their neighbors. This algorithm converges but the termination is implicit, nodes do not have a defined global termination criterion. Some nodes may temporarily consider themselves as belonging to the centroid. Although this works in most situations, this local election mechanism is, for instance, not sufficient to elect a single centroid node in torus-like networks (see Figure 3.1-d)) where all nodes are centroid and they are not all neighbors to each other. This algorithm uses $O(n)$ memory space per node (we ignore message storage cost), as each node has to store a list of already known minimal distances to the other nodes to handle cycles.

In [Lehmann et al., 2003], Lehmann et al. propose a distributed synchronous framework to compute the eccentricity, closeness and betweenness of all nodes. Initially, all nodes broadcast a message that contains its unique node identifier and the currently traveled number of hops. Every node receives back a report message that contains an id-pair (source and destination) along with the distance between them. To avoid circles, every node constructs a data structure of $O(n^2)$ id-pairs.

A distributed synchronous algorithm dedicated to the computation of the eccentricity of all nodes along with the network radius and diameter was proposed [Almeida et al., 2012].

It uses breadth-first search network traversals. Initially, all nodes initiate a BFS traversal that contains its unique node identifier and a hop counter. Nodes progressively learned the distance to all the others. Local criteria to detect convergence in each node, using the computed values and the number of consecutive rounds with no new BFS messages, are introduced. This algorithm requires the storage of $O(n)$ distances per node.

In [You et al., 2017], K. You et al. propose a distributed algorithm to compute the exact closeness centrality measures using only local interactions. At each round r, every node v_i sends its set of $(r-1)$-hop neighbors to all its (1-hop) neighbors. Upon reception of these messages by all neighbors, a node v_j can determine its set of r-hop neighbors using these messages and its own set of nodes within $r-1$ hops. This algorithm converges in $O(d)$ rounds but has a high storage cost per node as it ultimately requires the storage of $O(n)$ information. Moreover, the termination criterion relies on the knowledge of the diameter of the network, that can be distributively computed using the algorithm in [Garin et al., 2012].

As a consequence, existing asynchronous distributed algorithms designed to elect a node belonging to the exact center or centroid of arbitrary networks are not scalable. They involve a distributed APSP computation which has either a large time complexity or/and a large storage cost in systems composed of thousands of nodes with constrained computational power and restricted memory resources.

3.4.2/ Methods for Specific Classes of Graphs

Efficient heuristics have been proposed to compute the center and the centroid of tree graphs.

For instance, [Bruell et al., 1999, Patterson, 2014] propose distributed methods to compute the center of a tree graph in r rounds during which each node determines a sort of distance to the border of the network. The center is the set of nodes which have a greater value than all their neighbors. In [Bruell et al., 1999], the authors also propose an algorithm to compute the centroid of a tree graph in d rounds using a similar approach.

In [Handler, 1973], the authors propose Minimax, an efficient sequential algorithm to compute the center of undirected tree graphs using only two BFSes. It picks A, a random node of the tree, B the farthest node from A and C the farthest node from B. The center is at midpoint of path between B and C. However, in arbitrary graphs, Minimax does not always return the exact center. For example, in Figure 3.3, Minimax returns one of the module in the diagonal in blue.

Other sequential algorithms for specific classes of graphs include [Chepoi et al., 1994] for chordal graphs and [Lan et al., 1999] for weighted cactus graphs.

Although these approaches are efficient for the graphs they target, they are unfortunately not directly generalizable to arbitrary graphs.

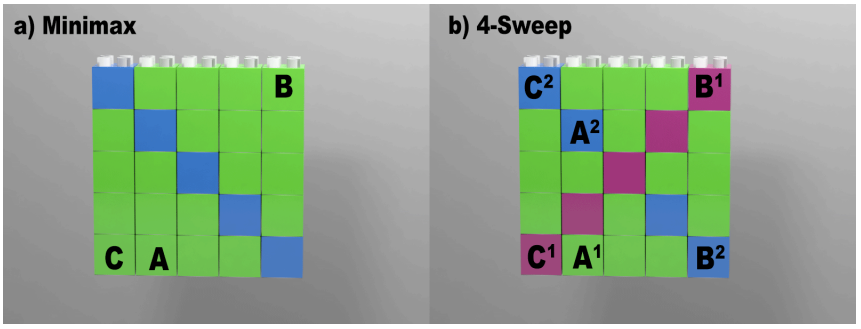


Figure 3.3: Minimax and 4-Sweep failure case.

3.4.3/ Sampling-based Methods

Sampling-based methods consist in computing a sampling of values (shortest paths, node degree, etc.). These algorithms fall into two main categories, namely the approaches that compute shortest paths from a sampling of nodes and the methods that use limited-scope value computations.

3.4.3.1/ Shortest Paths from a Sampling of Nodes

Some input-graph analysis approaches have recently been proposed in order to compute a central vertex of arbitrary graphs using a limited number of SSSP computations. Most of them use BFS computations.

In [Crescenzi et al., 2013], Crescenzi et al. propose the 4-Sweep sequential algorithm which finds a node with low eccentricity in arbitrary graphs using 4 BFSes. It essentially performs two consecutive Minimax. A^1, B^1 and C^1 are selected exactly as in Minimax. Then, A^2 is selected as a node at midpoint between B^1 and C^1. Repeating Minimax a second time, B^2 is one of the farthest nodes from A^2 and C^2 is one of the farthest nodes from B^2. 4-Sweep returns a node at mid-distance between B^2 and C^2. In the general case, the 4-Sweep algorithm does not return the exact center. For instance, in Figure 3.3, 4-Sweep returns one of the nodes on the two diagonals, depending on the position of A^1 and A^2 just as Minimax would have done.

In [Takes et al., 2013], the authors propose a sequential algorithm to compute the exact eccentricity of all nodes using a limited number of consecutive BFSes. This algorithm refines lower and upper bounds on the vertex eccentricities until convergence is reached.

In [Borassi et al., 2014], Borassi et al. propose a similar sequential algorithm to find the exact radius (i.e., the eccentricity of the center) and diameter. The algorithm stops earlier as all the eccentricities are not computed. The authors suggest that the algorithm should start by performing some BFSes from the least central vertices and propose the SumSweep approach. In this approach, the root of the next BFS is the node of minimum partial farness to the root of the previous BFSes, i.e., the nodes that maximize the sum to the roots of the previously performed BFSes. The complete algorithm may still require that a considerable number of BFSes should be performed, sometimes more than a hundred

in Blinky Blocks systems composed of 500 modules[4]. A distributed implementation of the complete algorithm to find an exact center is, thus, not an option as performing a hundred distributed BFSes in a consecutive way will take too much time.

In [Eppstein et al., 2001], the authors propose a sequential algorithm to estimate node closeness using partial closeness computation from a random sample of nodes. In [Dissler et al., 2016], the authors propose a distributed synchronous implementation of this algorithm. In this evaluation section, we show that performing BFSes from external nodes rather than from random ones leads to a better estimation of node centrality.

In [Roditty et al., 2013, Chechik et al., 2014], the authors propose input-graph analysis algorithms to estimate node eccentricity. These algorithms start to run some BFSes from a random sample of S nodes. Then they compute a BFS from node v, one of the farthest node to any another node in S, and from a certain number of the closest nodes from v. Finally, they derive some estimation of the node eccentricities.

A sequential framework to approximate node closeness and node betweenness was introduced in [Chan et al., 2009]. Closeness computations are performed on an abstract graph of small-diameter communities formed from tightly connected nodes.

All these existing approaches based on shortest paths computation from a sampling of nodes are promising but they do not fit our system model. They have been designed for input-graph analysis or target synchronous distributed systems.

3.4.3.2/ LIMITED-SCOPE CENTRALITY COMPUTATION

In Distributed Assessment of Network CEntrality (DANCE) [Wehmuth et al., 2011] and Distributed Assessment of the Closeness CEntrality Ranking (DACCER) [Wehmuth et al., 2013], every node computes its volume centrality, i.e., the sum of the node degree of the k-hop neighboring nodes, using $O(k)$ time and $O(|N^k_{v_i}|)$ memory per node v_i. The value of k impacts both the accuracy and the cost of these algorithms. k should be large enough to derive an accurate global centrality value from localized computations. DANCE and DACCER are best suited to networks that do not present a highly regular structure, have a small diameter compared to their size and have a low density. As shown in Section 2.2.5, LMRs do not exhibit these properties. In general, small-world (e.g., the Internet) or scale-free networks share such characteristics.

DANCE also builds a 3-level hierarchical structure that enables to locate both local and global central nodes. In large-scale systems, DANCE may have an important memory usage per node with regard to our strong storage restrictions for two reasons. Firstly, the number of neighbors at k hops may be important, e.g., in the Blinky Blocks system, $|N_{v_i}(k)| = O(k^3)$ (see Sections 2.2.5 and A.5.2). In terms of figures, the 2-hop neighborhood can be composed of up to 25 nodes, and the 3-hop neighborhood of up to 63 nodes. Secondly, the memory cost of the hierarchical structure can be important. Indeed, every node that exhibits a higher centrality value within a range of $(2k)$-hops, knows all the other

[4]Based on practical experiments realized using our implementation of this algorithm, which is available online at: https://github.com/nazandre/GraphAnalyzer

nodes that satisfy the same property. This knowledge is then used to identify the highest centrality node in the whole network.

3.4.4/ Probabilistic-Counter-based Methods

Algorithms based on low-memory-footprint probabilistic counters to estimate node centrality measures have recently been proposed in [Kang et al., 2011b, Kang et al., 2011a, Garin et al., 2012]. These algorithms are approximately equivalent to running a BFS from every node but at less expense in terms of computations and communications.The algorithms run in $O(d)$ rounds. At each round r, nodes estimate the size of their r-hop neighborhood using local interactions with their 1-hop neighbors. The averaged farness can be estimated using equation (3.6). The eccentricity of a node is either estimated using equation (3.2) or corresponds to the last round at which the internal state of the probabilistic counter has been updated.

In [Kang et al., 2011b, Kang et al., 2011a], the authors propose efficient input-graph analysis algorithms to respectively estimate the node averaged farness and eccentricity using the Flajolet-Martin probabilistic counter [Flajolet et al., 1985]. The algorithms terminate when no update has been performed for any node during a complete round. The internal state of a Flajolet-Martin counter is composed of k bitstrings of $O(\log n)$ bits, with $k \geq 1$ an input parameter. These algorithms run in $O(dm)$ time and store $O(dk)$ bitstrings of $O(\log n)$ bits instead of an array of $O(n)$ information per node in naive sequential approaches. In practice, these algorithms even require storage for only two counter states per node, i.e., one for the previous and current rounds. These algorithms have been evaluated on a distributed implementation based on the MapReduce programming model for large-scale and distributed data processing. However, these implementations still require a global view of the graph.

A synchronous distributed algorithm built to estimate node eccentricity in anonymous networks has been proposed in [Garin et al., 2012]. It uses a statistical network size estimation algorithm [Varagnolo et al., 2010] which is based on random number generations and a max-consensus procedure. This algorithm converges in $O(d)$ time and needs to store $O(k)$ random numbers per node. In [Garin et al., 2012], the algorithm assumes the number of rounds to be provided as input or computed using an external algorithm.

Interesting probabilistic counter-based algorithms have been proposed but they do not fit our assumptions.

3.4.5/ Other Approaches

3.4.5.1/ The Tree-based Centrality Measure

In [Kim et al., 2013], the authors propose the tree-based centrality measure. It uses only distance calculations in $T(v_r)$, a BFST of the network rooted at some random node v_r. The tree-based centrality of a node v_i is equal to a fixed programmer-defined centrality weight

w_c with $w_c > 1$, if the height of v_i in $T(v_r)$ is equal to the average distance to v_r. Otherwise, it is equal to 1. The authors suggest the combination of the tree-based centrality measure with other parameters, e.g., the remaining energy, to elect the more suitable node for a specific task. We name the set of nodes that maximize the tree-based centrality measure the tree-based center. Nodes with a height equal to the average distance to v_r belongs to that center. As shown in [Kim et al., 2013], the closeness centrality and the expected tree-based centrality over all possible choices of v_r and $T(v_r)$ have similar priorities. Therefore, we consider that the tree-based centrality is an approximation of the closeness centrality measure.

Electing a node in the tree-based center requires the election of an initiator and the construction of a single BFST rooted at this initiator. This algorithm runs in $O(d)$ time and $O(\Delta)$ memory space per node.

3.4.5.2/ Random Walks

An emergent approach to compute an approximate centroid of a distributed system is proposed in [Dutot et al., 2011]. A virtual ant colony explores the graph, virtually dropping pheromones on edges and nodes. The node that accumulates the largest amount of virtual pheromones is designated as the centroid. The main drawback of this method is that every ant must maintain a tabu list of visited nodes to handle cycles. This list is $O(n)$ memory space in the worst case. Moreover, the quality of the computed solution depends on the topology of the system. It performs well for trees but badly for grids.

3.4.5.3/ Linear Programming

In [Wang et al., 2015], the authors propose a scalable distributed algorithm to estimate node closeness centrality with only local interactions and a memory complexity per node of $O(\Delta)$. They define a regularized linear program based on the aggregation of a set of constraints that involves only nearby variables. A gradient algorithm is used to distributively solve this linear program over the network. The constraints of the linear program are augmented to its objective function as barriers and the algorithm converges progressively. An evaluation on networks with 6 to 50 nodes shows that this algorithm is on average 91% accurate in terms of closeness ordering. However, this algorithm has no well-established termination criterion that would be desirable to use the closeness values to elect an approximate centroid of the system.

3.4.6/ Summary

Table 3.2 summarizes the existing distributed algorithms. Computing exact center and centroid nodes in asynchronous distributed systems is an expensive operation in terms of messages and in terms of storage requirement and/or time. Algorithms designed for a specific class of graphs (e.g., tree graphs) are not generalizable to arbitrary graphs.

Algorithm	Type of center or, if none, centrality measure	Approach	Async. vs Sync.
[Korach et al., 1984]	center, centroid	exhaustive	async
[Lehmann et al., 2003]	closeness, eccentricity, betweenness	exhaustive	sync
[Almeida et al., 2012]	eccentricity	exhaustive	sync
[You et al., 2017]	closeness	exhaustive	UN
BARYCENTER [Mamei et al., 2005]	centroid	exhaustive	async
[Bruell et al., 1999]	centroid, center	Method for specific classes of graphs (tree)	UN
[Patterson, 2014]	center	Method for specific classes of graphs (tree)	sync
[Dissler et al., 2016]	closeness	Shortest path computations from a sampling of nodes	sync
DANCE [Wehmuth et al., 2011] DACCER [Wehmuth et al., 2013]	volume-based center*	Limited-scope centrality computation	async
[Garin et al., 2012]	eccentricity	Probabilistic-Counter-based method	async
[Dutot et al., 2011]	centroid	Random walks	async
[Wang et al., 2015]	closeness	Linear programming	async
[Kim et al., 2013]	tree-based center*	Computation on a tree	async

Our contributions:

k-BFS SumSweep	center, centroid	Shortest path computations from a sampling of nodes	async
ABC-Center	center	Shortest path computations from a sampling of nodes	async
PC2LE	center, centroid	Probabilistic-Counter-based method	async

Table 3.2: Summary of the state of the art on network centrality in distributed systems. If the algorithm comes with an election mechanism, we provide the type of the elected (approximate) central node. Otherwise, we give the name of the computed/estimated centrality measure. Note $*$: a specific low-complexity measure is proposed and used to elect a most central node. “UN” stands for “Unknown”.

Efficient sampling-based and probabilistic-counter-based methods have been proposed but they have not been applied to distributed asynchronous systems so far.

In this chapter, we propose asynchronous distributed algorithms to elect approximate-centroid and approximate-center nodes, namely ABC-Center, k-BFS SumSweep and Probabilistic-Counter-based Central-Leader Election (PC2LE). k-BFS SumSweep and ABC-Center are sample-based algorithms, i.e., they perform distributed BFSes from a sample of nodes, while PC2LE use probabilistic counting.

k-BFS SumSweep is based on the sequential SumSweep heuristic [Borassi et al., 2014]. ABC-Center extends the sequential Minimax [Handler, 1973] and 4-Sweep [Crescenzi et al., 2013] algorithms. PC2LE is inspired by the input-graph analysis algorithms [Kang et al., 2011a, Kang et al., 2011b] and the distributed synchronous algorithm [Garin et al., 2012]. PC2LE differs from these approaches. First of all, PC2LE targets asynchronous distributed systems. Secondly, PC2LE uses its own mechanism to estimate the diameter of the system in order to bound the number of computation rounds. Thirdly, any probabilistic counter can be used in PC2LE and we have experimentally observed that, for similar resource usage, the HyperLogLog counter [Flajolet et al., 2007] leads to more accuracy than the counters used in [Kang et al., 2011a, Kang et al., 2011b, Garin et al., 2012]. Finally, PC2LE comes with an election procedure to elect the most central node.

3.5/ Preliminary Materials on Network Traversal and Tree Algorithms

This section presents the primitives used to design our algorithms.

3.5.1/ Breadth-First Network Traversal and Spanning-Tree Construction

Our centrality-based leader election algorithms are all based on BFST constructions, which are used to compute distances and build paths to particular nodes. Constructing a BFST enables one to solve the SSSP problem since the distance from a node to the root in the tree corresponds to the distance from that node to the root in the complete network.

3.5.1.1/ Algorithm Choice

Building a spanning tree involves a time-communication trade-off [Awerbuch et al., 1985]. It requires at least $\Omega(d)$ time and $\Omega(m)$ messages [Awerbuch et al., 1985]. Different algorithms have been proposed for asynchronous systems.

Some of our algorithms (ABC-CenterV2 and k-BFS SumSweep) consecutively build a dozen or more BFSTs. It is crucial to use an algorithm that builds such a tree quickly in order to ensure an acceptable time of convergence for these algorithms. To the best of our knowledge, only Cheung's algorithm [Cheung, 1983] and Aspnes' one [Aspnes, 2017] run in $O(d)$ time (time due to message pileups is ignored here). Both of them use $O(\Delta)$ memory space at the application layer to store their neighbor states. Cheung's algorithm uses $O(nm)$ messages [Lynch, 1996] while Aspnes' algorithm uses $O(dm)$ messages.

CHEUNG-BFS-ST refers to Cheung's algorithm. Aspnes' algorithm does not have a global termination (i.e., the root of the tree does not know when the tree construction is finished). Our algorithms require the detection of the global termination of the tree

construction in order to continue their execution. In [Boulinier et al., 2008], the authors show how to enhance Aspnes' algorithm with a global termination criterion. ASPNES-BFS-ST-T refers to the Aspnes' algorithm combined with that global termination detection method.

Figures 3.4 and 3.5 respectively show the simulated execution time and the number of messages used by the CHEUNG-BFS-ST and the ASPNES-BFS-ST-T algorithms to construct a BFST on random Blinky Blocks systems. The tree construction is initiated by the root node and the other nodes locally start to execute the algorithm upon reception of the first message. It appears Cheung's algorithm performs better, both in terms of time and messages than Aspnes' algorithm in our experimental setup. Thus, we decide to use Cheung's algorithm with an optimization explained in the next section.

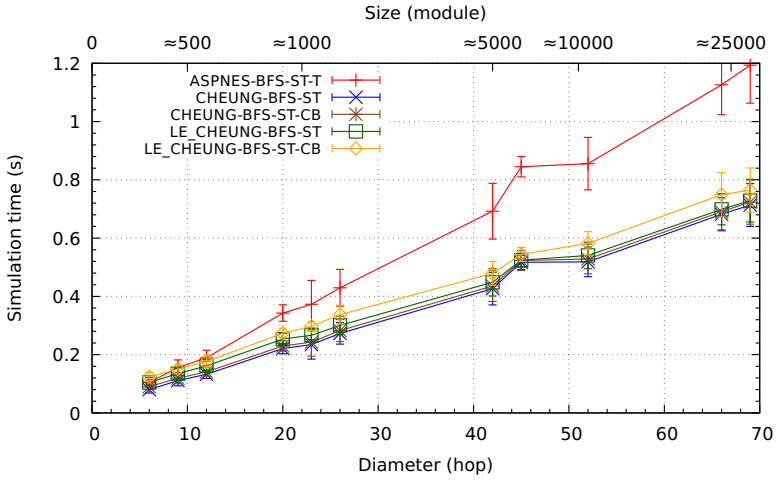


Figure 3.4: Simulated execution time of the BFST construction and leader election algorithms.

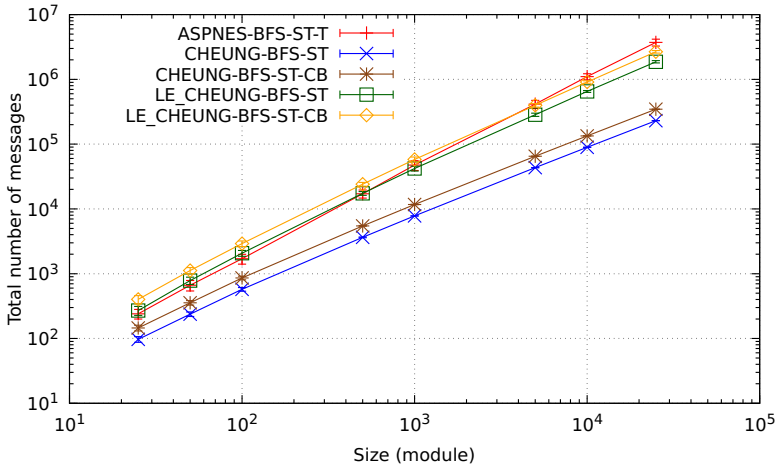


Figure 3.5: Total number of messages sent during the execution of the BFST construction and leader election algorithms.

3.5.1.2/ THE CONTROLLED-BROADCAST OPTIMIZATION

CHEUNG-BFS-ST adopts an echo approach. Initially, the root of the tree under construction starts a network traversal by sending to all its immediate neighbors a BFS_GO message which contains a hop counter initialized to 0. When a node receives a BFS_GO message with a smaller hop counter than the previously known one, then it forgets about the previous network traversal and starts participating in the new one. BFS_BACK messages are progressively sent back to the root node from the leaf nodes. The graph traversal terminates as soon as the root node gets notified by all its neighbors through BFS_BACK messages.

As stated in [Lynch, 1996], messages might pile up, which increases the execution time and the memory space usage. A module v_i might, for instance, receive up to $|N^1_{v_i}| - 1$ increasingly "better" BFS_GO messages in such a short amount of time from all of its neighbors except v_j and consequently insert $(|N^1_{v_i}|-1)$ BFS_GO messages in the outgoing-message queue dedicated to v_j in a so short amount of time that v_i is not able to completely send a message to v_j. This outgoing-message queue keeps growing if this situation happens several times. Furthermore, if it happens many times, at many nodes, the system gets congested.

To avoid this situation, we propose the controlled-broadcast optimization. This optimization is inspired by [Gallager, 1982] where the author's suggestion is to send in priority BFS_GO messages with a lower hop counter. In the controlled-broadcast optimization, a single BFS_GO message is present in any outgoing-message queue at a time. If the firmware/operating system allows the modification of a queued message, we propose to update the currently queued BFS_GO message (if there is any) rather than insert a new one in the queue. Otherwise, a node sends a new BFS_GO message to a neighboring node only if the previous BFS_GO message has been completely sent and removed from the outgoing-message queue. If the operating system enables to know when a message has been sent, this solution comes for free. Otherwise, nodes use an extra message to acknowledge every BFS_GO message they receive and a node does not send a new BFS_GO message until the previous one has been acknowledged. Note that this solution is not suitable in dense networks with shared communication medium where acknowledgment messages may cause many collisions.

We call Cheung's algorithm combined with the controlled-broadcast optimization the CHEUNG-BFS-ST-CB algorithm. The pseudo-code of CHEUNG-BFS-ST-CB is given in Algorithm 1.

The controlled-broadcast optimization ensures that only $O(1)$ messages are present in any outgoing-message queue at a time. Thus, it prevents message pileups. It follows that there are at most $O(\Delta)$ messages at each node. Moreover, the algorithm variable memory usage is $O(\Delta)$. Hence, the total memory usage of CHEUNG-BFS-ST-CB is $O(\Delta)$. Moreover, the controlled-broadcast assumption does not change the asymptotic complexities (where pileups are ignored). The first two solutions come for free. In the third case, every BFS_GO message is acknowledged, thus at most $O(2nm) = O(nm)$ messages are sent. Moreover, since there are at most $O(1)$ messages in every single outgoing-message

queue, the best BFS_GO message propagates through the network in $O(d)$ time. In this work, we use the third solution as it is the most general one, even though it is the most expensive one.

As shown in Figures 3.6 and 3.7, the controlled-broadcast optimization has no perceptible benefit on the maximum queue occupancy and the maximum memory usage of Cheung's algorithm in our experimental setup. This is because our optimization prevents a worst-case issue that rarely occurs in our target systems where the network is unloaded and all the links are configured with the same bitrate. However, our optimization has a significant impact on the maximum memory usage of the leader election algorithm based on Cheung's algorithm presented in the next section. Also note that the controlled-broadcast optimization only generates a low time and message overhead (see Figures 3.4 and 3.5).

3.5.2/ LEADER ELECTION BASED ON NETWORK TRAVERSAL ALGORITHMS

Our centrality-based leader elections all start by electing an initiator.

Network traversal algorithms can be used to elect a leader [Raynal, 2013]. All nodes initiate concurrent parallel network traversals and a single one terminates. The node that initiates this traversal becomes the initiator. We call the leader election algorithm

Variants : CHEUNG-BFS-ST-CB // Black lines only
⌞LE_CHEUNG-BFS-ST-CB⌝ // Black + ⌞---⌝ lines
†CHEUNG-BFS-ST-CB-AGG† // Black + †---† lines

Input : $N^1_{v_i}$ // v_i's 1-hop neighborhood
id_{v_i}// unique identifier of v_i
†*size*† // network size
// handler functions:
†*handleAppData*, *resetAppAggs*, *updateAppAggs* and *getAppAggs*†

Output : // Constructed tree (composed of v_i's parent and v_i's children):
tree<*parent*, *Children*>
distance // Distance of v_i to the root of the tree
†*data*†// Ordered list of data propagated from the root
†*aggregates*†// Ordered list of aggregates computed on v_i's subtree

1 **Initialization** of node v_i:
2 $finished \leftarrow false$; $tree.parent \leftarrow \perp$; $tree.Children \leftarrow \emptyset$; $Wait \leftarrow \emptyset$;
3 **if** v_i *root of the tree* **then** // (true for all nodes in LE_CHEUNG-BFS-ST-CB)
4 $distance \leftarrow 0$; $id \leftarrow id_{v_i}$;
5 **else**
6 $distance \leftarrow +\infty$; $id \leftarrow \perp$;
7 † $data \leftarrow \emptyset$; $branchSize \leftarrow \emptyset$; $resetAppAggs()$; $aggregates \leftarrow <1> \cup getAppAggs()$; †
8 **When** Algorithm variant **starts** at node v_i **do**:
9 // Executed by the root node only
for each $v_j \in N^1_{v_i}$ **do**
10 **send** BFS_GO<id, $distance$,†$data$†> **to** v_j;
11 $Wait \leftarrow Wait \cup \{v_j\}$;
12 **if** $Wait = \emptyset$ **then**
13 Algorithm variant **terminates**;

```
14 When BFS_GO(mid, dist, †dat†) is received by v_i from v_j do:
15   † data ← dat; handleAppData(); †
16   if ⌞(mid < id)⌝ OR id = ⊥ then
17     id ← mid; distance ← +∞; tree.parent ← ⊥;
18   if (id = mid) AND (dist + 1 < distance) then
19     if (tree.parent ≠ ⊥) then
20       send BFS_BACK<id, distance − 1, false, †{}†> to tree.parent;
21     tree.parent ← v_j; tree.Children ← ∅; distance ← dist + 1; Wait ← ∅;
22     † branchSize ← ∅;
23     resetAppAggs(); †
24     for each v_k ∈ N^1_{v_i} \ {tree.parent} do
25       send BFS_GO<id,distance, †data†> to v_k. /* Controlled-broadcast optimization (avoid
           congestion): send that message only if v_i's outgoing queue to v_k does not
           contain any other BFS_GO message. Otherwise, wait until this message has
           been sent and then send the best known <id, distance> value(s) at that future
           time in a BFS_GO message                                                    */
26       Wait ← Wait ∪ {v_k};
27     if Wait = ∅ then
28       s ← 1; aggregates ← <s> ∪ getAppAggs();
29       send BFS_BACK<id, distance − 1, false, †aggregates†> to tree.parent;
30   else if id = mid then
31     send BFS_BACK<mid,dist, false,{}> to v_j;

32 When BFS_BACK(mid, dist, c, †aggs†) is received by v_i from v_j do:
33   if (id = mid) AND (distance = dist) AND ¬ finished then
34     Wait ← Wait − {v_j};
35     if c = true then
36       tree.Children ← tree.Children ∪ v_j;
37       † branchSize[v_j] ← aggs[0]†;
38     else
39       tree.Children ← tree.Children − v_j;
40       † remove branchSize[v_j]†;
41     † appAggUpdate(v_j, c, aggs); †
42     if Wait = ∅ then
43       † s ← 1 + ∑_{v_k} branchSize[v_k]; †
44       if tree.parent = ⊥ then
45         finished ← true;
46         † if s ≠ size then
47           // Wait (aggregate values may be uncorrect)
48           finished ← false;
49           return;
50         aggregates ← <size> ∪ getAppAggs(); †
51         Algorithm variant terminates;
52       else
53         † aggregates ← <s> ∪ getAppAggs(); †
54         send BFS_BACK<id, distance − 1, true, †aggregates†> to tree.parent;
```

Algorithm 1: Pseudo-code for any code v_i of different algorithms based on Cheung's BFST algorithm: CHEUNG-BFS-ST-CB, LE_CHEUNG-BFS-ST-CB and CHEUNG-BFS-ST-CB-AGG.

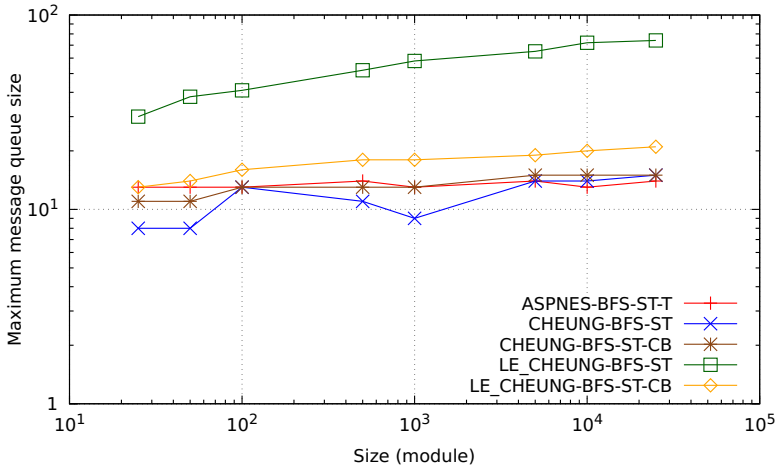


Figure 3.6: Maximum queue length of the BFST construction and leader election algorithms.

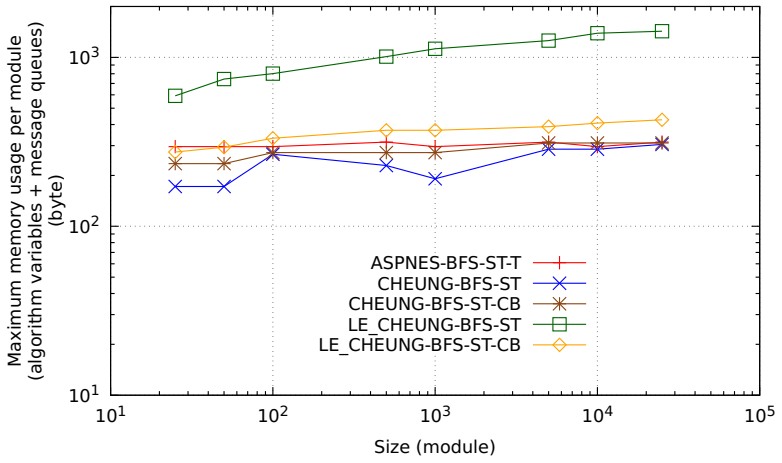


Figure 3.7: Maximum memory usage of the BFST construction and leader election algorithms. Memory usage takes into account both the algorithm variables and the messages in the queues.

based on the CHEUNG-BFS-ST algorithm the LE_CHEUNG-BFS-ST. At the end of LE_CHEUNG-BFS-ST, an initiator has been elected and a BFST rooted at this node has been constructed. In addition, all nodes know their distance to the root.

LE_CHEUNG-BFS-ST-CB refers to the LE_CHEUNG-BFS-ST combined with the controlled-broadcast optimization. Algorithm 1 provides the pseudo-code of LE_CHEUNG-BFS-ST-CB. In LE_CHEUNG-BFS-ST-CB, nodes initiate n concurrent CHEUNG-BFS-ST-CB. Thus, LE_CHEUNG-BFS-ST-CB runs in $O(d)$ and uses $O(mn^2)$ messages and $O(\Delta)$ memory space per node.

We compare LE_CHEUNG-BFS-ST and LE_CHEUNG-BFS-ST-CB to show the importance of the controlled-broadcast optimization in the leader election algorithm. As shown in Figure 3.6, LE_CHEUNG-BFS-ST-CB has a significantly lower maximum queue occupancy than LE_CHEUNG-BFS-ST. Hence, the controlled-broadcast optimization greatly

reduces the maximum memory usage (see Figure 3.7), while it only generates a low time and message overhead (see Figures 3.4 and 3.5).

3.5.3/ Broadcast and Convergecast on a Spanning Tree

The Spanning-Tree Broadcast (STB) and Spanning-Tree Convergecast (STC) algorithms are two fundamental primitives used in distributed algorithms [Lynch, 1996, Raynal, 2013].

In STB, some data are propagated down from the root of the spanning tree to all the nodes of the system along the edges of the tree. Algorithm 2 shows the pseudo-code of STB.

```
   Variants      : STB // Black lines only
                   ⌊STB-STC⌉ // Black + ⌊___⌉ lines
   Input         : tree<parent, Children> // tree: v_i's parent and Children
                   handleAppData // handler function
                   ⌊updateAppAggs⌉ // handler function
   Output        : data// Ordered list of data propagated from the root
                   aggregates// Ordered list of aggregates computed over v_i's subtree
   Primitive(s)  : ⌊STC(tree : tree, handlers : updateAppAggs)⌉ // see Algorithm 3

1  Initialization of node v_i:
2  data ← ∅;

3  When STB starts at root node v_i do:
4  for each v_j ∈ tree.Children do
5      send STB_GO(data) to v_j;

6  When STB_GO<dat> is received by v_i from v_j do:
7  data ← dat;
8  handleAppData();
9  ⌊initialize STC; start STC;⌉
10 for each v_j ∈ tree.Children do
11     send STB_GO<data> to v_j;
```

Algorithm 2: Pseudo-code for any code v_i of the Spanning-Tree Broadcast (STB) algorithm with data propagation and the STB-STC (Spanning-Tree Broadcast followed by Spanning-Tree Convergecast) algorithm.

The STC algorithm is the inverse of STB. In STC, a convergecast message is forwarded back from the leaves to the root of the tree. Leaves start to send a convergecast message to their parent. Inner nodes wait until they have received a convergecast message from all their children before sending a convergecast message to their respective parent. The algorithm terminates once the root of the tree has received a convergecast message from all its neighbors. STC can be triggered by an STB (see Algorithm 2, line 9). STB-STC refers to this combination of the execution of STB immediately followed by the execution of STC. Algorithm 3 shows the pseudo-code of STC.

STB, STC and STB-STC use $O(d)$ time, $O(n)$ messages and require $O(\Delta)$ memory space per node, if we assume that the propagated and computed data can be stored using

$O(1)$ memory space per node. These algorithms use only $O(1)$ variables, but they have a memory complexity of $O(\Delta)$ due to the storage cost of both the input spanning tree and the messages.

```
   Input  : tree<parent, Children> // tree: v_i's parent and Children
            updateAppAggs // handler function
   Output : aggregates// Ordered list of aggregates computed over v_i's subtree

1  Initialization of node v_i:
2  aggregates ← <>; waiting ← |tree.Children|;

3  When STC starts at node v_i do:
4  if waiting = 0 then
5      if tree.parent =⊥ then
6          STC terminates;
7      else
8          send STC_BACK<aggregates> to tree.parent;

9  When STC_BACK<aggs> is received by node v_i from v_j do:
10 waiting ← waiting − 1;
11 updateAppAggs(v_j, aggs);
12 if waiting = 0 then
13     if tree.parent =⊥ then
14         STC terminates;
15     else
16         send STC_BACK<aggregates> to tree.parent;
```

Algorithm 3: Pseudo-code for any code v_i of the Spanning-Tree Convergecast (STC) algorithm with aggregate computation.

3.5.4/ Global Data Diffusion and Global-Aggregate Computation

STB can be used to globally spread some information to all nodes in the system. STC can be used to compute network-wide aggregates [Lynch, 1996, Raynal, 2013] (e.g., the number of nodes in the system, the maximum distance to the root node, i.e., the height of the tree, the next hop on the path to a node that minimizes/maximizes a specific value, etc.). In this chapter, propagated data and aggregates are assumed to be stored using $O(1)$ memory space per node.

CHEUNG-BFS-ST-CB can also be used to spread information to all nodes in the network and to compute aggregates during the construction of a spanning-tree. CHEUNG-BFS-ST-CB-AGG refers to the execution of CHEUNG-BFS-ST-CB during which some data is spread through the system and some aggregates are computed. The pseudo-code of CHEUNG-BFS-ST-CB-AGG is provided in Algorithm 1. The information to be spread to all nodes is directly attached with the BFS_GO messages. Computing aggregates about the tree (e.g., its height, path to the farthest node, etc.) during its construction is more tricky, as CHEUNG-BFS-ST-CB does not have a local termination criterion, i.e., a non-root node does not know when its involvement in the tree construction process is finished. Indeed, a node may finally leave a subtree for a better one at any time [Raynal, 2013].

Hence, a node maintains aggregate values of all its subtree branches and these partial aggregates are updated whenever a change occurs in its subtree (see Algorithm 1, lines 37, 40 and 41). Moreover, it may happen that a leaf node finally leaves a subtree, whereas its previous parent has already initiated the BFS_BACK wave, with a possibly wrong aggregate value, toward the root of the tree. The previous parent then initiates a second BFS_BACK wave. This second wave, with the correct aggregated branch value, might possibly arrive anytime after the root has received a BFS_BACK message from all its other branches (i.e., after having detected the termination of the tree construction and possibly after having launched some other processes which use the wrong computed value). To ensure that the last BFS_BACK message has been received, we check that all nodes have participated in the aggregate computation only once. In order to do it, CHEUNG-BFS-ST-CB maintains at the root node a counter of the number of nodes that have participated in the aggregate computation. The value of this counter is compared to the actual network size which must be provided as input. The root node knows that the aggregate has been properly computed when these two values are equal (see Algorithm 1, line 47).

We do not assume that the network size is known at the system startup. In practice, our centrality-based leader election algorithms first elect an initiator using LE_CHEUNG-BFS-ST-CB and then perform an STB-STC to compute the network size and other aggregates. LE_CHEUNG-BFS-ST-CB_STB-STC refers to this procedure. The pseudo-code of LE_CHEUNG-BFS-ST-CB_STB-STC is given in Algorithm 4. The value of the network size can then be used to control the execution of CHEUNG-BFS-ST-CB.

```
   Primitive(s)        : LE_CHEUNG-BFS-ST-CB
                         STB-STC(tree : LE_CHEUNG-BFS-ST-CB.tree, handlers :⊥, stcHandler)

    // Initialization and start handlers:
 1  Initialization of v_i:
 2  size ← 1; height ← 0; nextHopToFarthest ←⊥;
 3  STB-STC.aggregates ← {size, height};

 4  When LE_CHEUNG-BFS-ST-CB_STB-STC starts at node v_i do:
 5  start LE_CHEUNG-BFS-ST-CB;

    // Primitive handlers for aggregate computation and data propagation:
 6  Function stcHandler(source, aggs):
 7      size ← size + aggs[0];
 8      if height < aggs[1] + 1 then
 9          height ← aggs[1] + 1; nextHopToFarthest ← source;
10      STB-STC.aggregates ← <size, height>;

    // Primitive termination handlers:
11  When LE_CHEUNG-BFS-ST-CB terminates at root node v_i do:
12  start STB-STC;

13  When STB-STC terminates at root node v_i do:
14  LE_CHEUNG-BFS-ST-CB_STB-STC terminates;
```

Algorithm 4: LE_CHEUNG-BFS-ST-CB_STB-STC detailed for any node v_i.

3.5.5/ Robustness to Module Mobility and Faults

To handle dynamic topology changes due to module mobility or to failure, a node launches a new central node election upon detection of a neighbor arrival or departure. Any local change may indeed have drastically changed the global topology of the network. Our algorithms are designed for fairly static networks where faults and node mobility only occur occasionally.

We use the technique proposed in [Vasudevan et al., 2004]. Each node participates in only one central node election at a time. In order to achieve this, an election index is used. This election index is a pair $<e, id>$ where id is the identifier of the node that has initiated the election and e is a number that is locally incremented each time a node triggers a new election. id is used to break the tie among concurrent elections with the same e value. A total order is defined on the election indices to determine election priority: $<e_1, id_1>$ has a higher priority than $<e_2, id_2>$ if $e_1 > e_2$ or if $e_1 = e_2$ and $id_1 < id_2$. Whenever a module receives a message for an election with a higher priority, it starts participating in this election and stops participating in any potential ongoing election of lower priority.

Note that this mechanism is used as an underlying transparent service and does not appear in the description of our algorithms.

3.5.6/ Summary of the Primitives and Notation

Table 3.3 summarizes the properties of the different primitives used to build our centrality-based leader election algorithms.

Algorithm	Description	Complexity		
		Memory space	Time	# Messages
LE_CHEUNG-BFS-ST-CB_STB-STC	Elect the minimum-id node as a leader and construct a BFST rooted at it. Then, perform a broadcast/convergecast to compute the size of the network along with height of the tree and a path to the farthest node (see Section 3.5.4).	$O(\Delta)^*$	$O(d)$	$O(mn^2)$
CHEUNG-BFS-ST-CB($tree, handlers$)	Construct a BFST rooted at the initiator with data propagation and aggregate computation using the handler functions (see sections 3.5.1 and 3.5.4).	$O(\Delta)^*$	$O(d)$	$O(nm)$
STB($tree, handler$)	Broadcast on tree with data propagation. Data is handled using the input handler function (see sections 3.5.3 and 3.5.4).	$O(\Delta)^*$	$O(d)$	$O(n)$
STC($tree, handler$)	Convergecast on tree with aggregate computation using the input handler function (see sections 3.5.3 and 3.5.4).	$O(\Delta)^*$	$O(d)$	$O(n)$
STB-STC($tree, handlers$)	Broadcast, then convergecast on tree with data propagation and aggregate computation using the input handler functions (see sections 3.5.3 and 3.5.4).	$O(\Delta)^*$	$O(d)$	$O(n)$

Table 3.3: Primitives used to build our centrality-based leader election algorithms. Note *: in memory complexity calculation it is assumed that propagated and computed data can be stored using $O(1)$ memory space.

3.6/ k-BFS SumSweep Framework

The k-BFS SumSweep framework elects either an approximate center or an approximate centroid node of the system. We first describe the general idea of our framework. Then we provide a detailed description of its distributed implementation. Afterwards, we analyze the complexity of that implementation.

3.6.1/ Description at a Glance

The k-BFS SumSweep framework is based on the SumSweep heuristic proposed as a starting point of the sequential algorithm in [Borassi et al., 2014] to compute the exact graph diameter and radius. SumSweep aims at consecutively selecting the most external

vertices of a graph. Our distributed implementation of k-BFS SumSweep uses distributed BFSes to compute SSSP.

In our framework, a partial centrality value (eccentricity or farness, depending on the framework version) is computed for every node using distances to $\{u^\lambda\}_{1\leq\lambda\leq k} \subseteq V$, a subset of k nodes, with $k \leq n$. This subset is formed from a random initial vertex and $k-1$ external vertices selected in a consecutive manner. The main idea behind our framework is that central nodes are first and foremost central to the most external nodes.

For pedagogical purposes, a sequential version of the k-BFS SumSweep framework is shown in Algorithm 5. Our framework runs in at most k iterations. During each iteration λ, a node u^λ is selected and the partial centrality value of every node is updated using the distance to u^λ (line 5-13). $\{u^\lambda\}_{1<\lambda\leq k}$ are some of the most external nodes consecutively selected using the SumSweep heuristic, i.e., the next vertex is the vertex of maximum partial farness that has not been previously selected (line 6). Note than u^1 is selected at random as initially all partial farness values are null. In our distributed implementation, the node of minimum identifier is elected as u^1. At the end, the node of minimum partial eccentricity (resp. farness) is elected as the approximate center (resp. centroid) of the system (line 14).

```
   Input  : G = (V, E) // network representation
            version ∈ {center, centroid}
            k // Number of nodes to select
   Output : centrality[ ] // approximate eccentricity or farness of every node
            central // approximate center or centroid of the system
1  for each v_i ∈ V do
2      far[v_i] ← 0;
3      centrality[v_i] ← 0;
4  Candidates ← V;
5  for λ = 1 to min{k, n} do
6      u ← v_i ∈ argmax_{v_j ∈ Candidates} far[v_j]; // SumSweep heuristic (ties are broken arbitrarily)
7      Candidates ← V − {u};
8      for each v_i ∈ V do
9          far[v_i] ← far[v_i] + d(u, v_i);
10         if version = center then
11             centrality[v_i] ← max{centrality[v_i], d(u, v_i)};
12         else
13             centrality[v_i] ← far[v_i];
14 central ← v_i ∈ argmin_{v_j ∈ V} centrality[v_j]; // a node of minimum centrality value is elected
```

Algorithm 5: Sequential version of k-BFS SumSweep framework.

Figure 3.8 depicts an execution of the k-BFS SumSweep framework on a 200-node Blinky Blocks system with $k = 10$. For both the center and centroid versions, the elected node is close to the theoretical node. In the evaluation section, we show that $k = 10$ provides accurate results even with large-scale systems of 10^4 nodes.

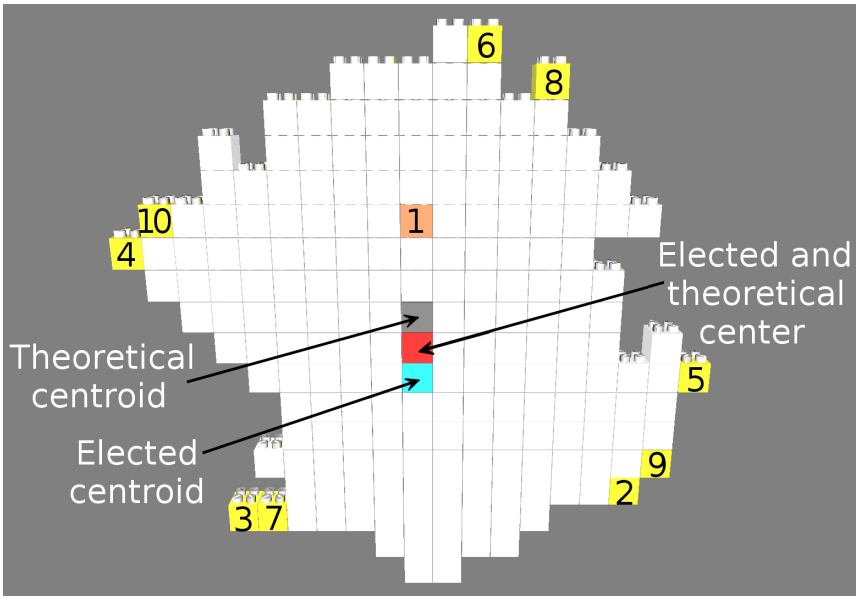


Figure 3.8: The k-BFS SumSweep framework running on a random two-dimensional Blinky Blocks system composed of 200 modules with $k = 10$. The initial module from which is performed the first distance computation is in brown. The other $k - 1$ external nodes selected are in yellow and the order of selection is written on them. In the center version of our framework, the module in red is elected and it matches the theoretical center. In the centroid version, the module in cyan is elected while the exact centroid is the (nearby) module in grey.

3.6.2/ Distributed Implementation

Algorithm 6 provides the pseudo-code of our distributed implementation of the k-BFS SumSweep framework. It uses two of our primitives (LE_CHEUNG-BFS-ST-CB_STB-STC and CHEUNG-BFS-ST-CB-AGG) and two specific types of messages (NEXT_BFS and ELECTED).

We recall that our framework runs in at most k iterations. During each iteration λ, a node u^λ is selected and the partial centrality value of every node is updated using the distance to u^λ. u^1 is elected using LE_CHEUNG-BFS-ST-CB_STB-STC (lines 5 and 47). If both $k > 1$ and $n > 1$, then a NEXT_BFS message is sent toward u^2, the farthest node from that initiator (lines 50-51 and 63-75). Otherwise, u^1 is elected as the central node and k-BFS SumSweep terminates (line 53).

Every iteration $\lambda > 1$ starts when u^λ receives a NEXT_BFS message. Upon reception of that message, u^λ initiates a CHEUNG-BFS-ST-CB-AGG (lines 63-75). CHEUNG-BFS-ST-CB-AGG is used to compute the node distance to u^λ and to construct both a path to a candidate node of maximum partial farness and a path to a node of minimum centrality value (lines 30-46). Upon termination of CHEUNG-BFS-ST-CB-AGG, u^λ sends a NEXT_BFS message toward $u^{\lambda+1}$ in order to trigger a next iteration if $k > \lambda$ and $\lambda > n$ (line 62). Otherwise, u^λ elects the node of minimum partial centrality value. If u^λ has the minimum centrality value, k-BFS SumSweep terminates and u^λ is elected as the central

node (line 58). Otherwise, u^λ sends an ELECTED messsage toward the node of minimum centrality value (lines 60 and 76-80). Upon reception of that message by the node of minimum centrality value, our framework terminates and this node is elected as the central node (line 78).

Note that, because the distance to u^λ may change several times during the execution of the CHEUNG-BFS-ST-CB-AGG launched by u^λ, the partial farness and centrality are actually updated only after the execution of CHEUNG-BFS-ST-CB-AGG triggered by $u^{\lambda+1}$ has started (see lines 6-14, 19 and 69). Hence, upon termination of the k-BFS SumSweep framework with $k > 1$, the elected node is the node of minimum partial centrality value computed over all the selected nodes, but the actual value in the variable *centrality* of all

Input : $version \in \{center, centroid\}$
k // Number of BFSes to perform
Output : a single central node is elected
Primitive(s) : LE_CHEUNG-BFS-ST-CB_STB-STC
CHEUNG-BFS-ST-CB-AGG($handlers$: $handleBFSData, updateBFSAggs, getBFSAggs, resetBFSAggs$)

// Initialization and start handlers:
1 **Initialization** of v_i:
2 $candidate \leftarrow true$; $iteration \leftarrow 0$; $far \leftarrow 0$ $centrality \leftarrow 0$; $branchCentrality \leftarrow \{\}$; $branchFarCandidate \leftarrow \{\}$; $nextHopToMinCentrality \leftarrow \perp$; $nextHopToMaxFarCandidate \leftarrow \perp$;
3 **start** k-BFS SumSweep;

4 **When** k-BFS SumSweep **starts** at node v_i **do**:
5 **start** LE_CHEUNG-BFS-ST-CB_STB-STC;

// Helper functions:
6 **Function** *updateLocalValues()* **:**
7 $dist \leftarrow$ CHEUNG-BFS-ST-CB-AGG.$distance$;
8 **if** $iteration = 1$ **then**
9 $dist \leftarrow$ LE_CHEUNG-BFS-ST-CB_STB-STC.$distance$;
10 $far \leftarrow far + dist$;
11 **if** $version = center$ **then**
12 $centrality \leftarrow \max\{centrality, dist\}$;
13 **else**
14 $centrality \leftarrow far$;

// Primitive handlers for aggregate computation and data propagation:
15 **Function** *handleBFSData()*:
16 $iter \leftarrow$ CHEUNG-BFS-ST-CB-AGG.$data[0]$;
17 **if** $iter > iteration$ **then**
18 $iteration \leftarrow iter$;
19 updateLocalValues() ;
// Take part in this BFS as non-root:
20 **re-initialize** CHEUNG-BFS-ST-CB-AGG;
21 **Function** *resetBFSAggs()*:
22 $branchCentrality \leftarrow \{\}$; $branchFarCandidate \leftarrow \{\}$;
23 **Function** *updateBFSAggs*($v_j, child, aggs$) **:**
24 **if** $child = true$ **then**
25 $branchCentrality[v_j] = aggs[1]$;
26 $branchFarCandidate[v_j] = aggs[2]$;
27 **else**
28 **remove** $branchFarCandidate[v_j]$;
29 **remove** $branchCentrality[v_j]$;
30 **Function** *getBFSAggs()*:
31 $dist \leftarrow$ CHEUNG-BFS-ST-CB-AGG.$distance$;
// Maximum candidate farness:
32 $maxCandidateFar \leftarrow 0$; $nextHopToMaxFarCandidate \leftarrow \perp$;
33 **if** $candidate = true$ **then**
34 $maxFar \leftarrow far + dist$;
35 $v_f \leftarrow \text{argmax}_{v_k \in N^1_{v_i}} branchFarCandidate[v_k]$;
36 **if** $v_f \neq \perp$ *AND* $branchFarCandidate[v_f] > maxFar$ **then**
37 $maxCandidateFar \leftarrow branchFarCandidate[v_f]$; $nextHopToMaxFarCandidate \leftarrow v_f$;
// Minimum centrality value:
38 **if** $version = center$ **then**
39 $minCentrality \leftarrow \max\{centrality, dist\}$;
40 **else**
41 $minCentrality \leftarrow centrality + dist$;
42 $nextHopToMinCentrality \leftarrow \perp$;
43 $v_f \leftarrow \text{argmax}_{v_k \in N^1_{v_i}} branchMinCentrality[v_k]$;
44 **if** $v_f \neq \perp$ *AND* $branchMinCentrality[v_f] < minCentrality$ **then**
45 $minCentrality \leftarrow branchMinCentrality[v_f]$; $nextHopToMinCentrality \leftarrow v_f$;
46 **return** $<maxFar, minCentrality>$;

```
   // Primitive termination handlers:
47 When LE_CHEUNG-BFS-ST-CB_STB-STC terminates
   at root node v_i do:
48 candidate ← false;
49 size ← LE_CHEUNG-BFS-ST-CB_STB-STC.size;
50 if size > 1 AND k > 1 then
51 |  send NEXT_BFS<size> to LE_CHEUNG-BFS-ST-
   |  CB_STB-STC.nextHopToFarthest;
52 else
53 |  k-BFS SumSweep terminates; // v_i is elected

54 When CHEUNG-BFS-ST-CB-AGG terminates at root
   node v_i do:
55 size ← CHEUNG-BFS-ST-CB-AGG.size;
56 if iteration + 1 = k OR iteration + 1 = size then
57 |  if nextHopToMinCentrality =⊥ then
58 |  |  k-BFS SumSweep terminates; // v_i is
   |  |     elected
59 |  else
60 |  |  send ELECTED<> to nextHopToMinCentrality;
61 else
62 |  send NEXT_BFS<size> to
   |  nextHopToMaxFarCandidate;

   // k-BFS SumSweep message handlers:
63 When NEXT_BFS<size> message is received by the
   node v_i do:
64 pathNextBFS = nextHopToMaxFarCandidate;
65 if iteration = 0 then //
66 |  pathNextBFS ← LE_CHEUNG-BFS-ST-CB_STB-
   |  STC.nextHopToFarthest;
67 if pathNextBFS =⊥ then
68 |  iteration ← iteration + 1;
69 |  candidate ← false; updateLocalValues();
   |  // Start a new BFS as root:
70 |  re-initialize CHEUNG-BFS-ST-CB-AGG;
71 |  CHEUNG-BFS-ST-CB-AGG.size ← size;
72 |  CHEUNG-BFS-ST-CB-AGG.data[0] ← iteration;
73 |  start CHEUNG-BFS-ST-CB-AGG;
74 else
75 |  send NEXT_BFS<size> to pathNextBFS;

76 When ELECTED<> message is received by node v_i do:
77 if nextHopToMinCentrality =⊥ then
78 |  k-BFS SumSweep terminates; // v_i is elected
79 else
80 |  send ELECTED<> to nextHopToMinCentrality;
```

Algorithm 6: Distributed implementation of the k-BFS SumSweep framework detailed for any node v_i.

nodes does not take into account the distance to the last selected node.

3.6.3/ Termination Proof and Complexity Analysis

The k-BFS framework sequentially runs $1\times$ LE_CHEUNG-BFS-ST-CB_STB-STC, then $(k-1)\times$ CHEUNG-BFS-ST-CB-AGG and finally forwards an ELECTED message toward the node of minimum centrality value through the last constructed spanning-tree. This message reaches its final destination using $O(d)$ time and $O(d)$ messages. All these steps terminate, thus our framework terminates. Moreover, we have $k \leq n$. Using the primitive complexity given in Section 3.5, the k-BFS framework runs in $O(kd)$ time using $O(mn^2)$ messages and $O(\Delta)$ memory space per module.

3.7/ ABC-Center

This section presents the ABC-Center algorithm which elects an approximate center of the system. We have designed two versions of ABC-Center, namely ABC-CenterV1 and ABC-CenterV2. ABC-CenterV2 was designed later in time and improves our first version in terms of accuracy, communication efficiency, memory usage and execution time. We present both of them in this section.

3.7.1/ DESCRIPTION AT A GLANCE

ABC-Center extends the sequential Minimax [Handler, 1973] and 4-Sweep [Crescenzi et al., 2013] algorithms. The main idea of ABC-Center is that central nodes lie in the middle of a diameter path. ABC-Center identifies an extreme path and recursively isolates midpoints on it until electing a single node. In contrast to the k-BFS SumSweep, the termination of ABC-Center does not rely on an input parameter.

Variants : ABC-CenterV2 // Black lines only
ABC-CenterV1 // Black + lines
Input : $G = (V, E)$ // network representation
Output : *central* // a central node

1 $Candidates \leftarrow V$; // all nodes are initially candidate
2 **while** $|Candidates| > 2$ **do**
3 $A \leftarrow v_i \in Candidates$; // A is a random candidate node
4 $B \leftarrow v_i \in \underset{v_j \in Candidates}{\text{argmax}} \; d(A, v_j)$; // B is one of the farthest candidate node from A
5 $C \leftarrow v_i \in \underset{v_j \in Candidates}{\text{argmax}} \; d(B, v_j)$; // C is one of the farthest candidate node from B
// most equi-distant candidates from B and C remain candidate:
6 $Candidates \leftarrow \underset{v_j \in Candidates}{\text{argmin}} \; |d(B, v_j) - d(C, v_j)|$;
// B and C are eliminated (if not already purged by the previous line)
7 $Candidates \leftarrow Candidates - \{B, C\}$;
8 **if** $|Candidates| = 2$ **then**
// Final step: most equi-distant candidates on a shortest path from B to C remain candidate
9 $Candidates \leftarrow \underset{v_j \in Candidates}{\text{argmin}} \; max\{d(B, v_j), d(C, v_j)\}$;
10 $central \leftarrow v_i \in Candidates$; // a node that remains candidate is arbitrarily selected

Algorithm 7: Sequential versions of ABC-CenterV1 and ABC-CenterV2.

For pedagogical purposes, a sequential version of ABC-Center is shown in Algorithm 7. ABC-Center iteratively finds an approximate center of the system. At the beginning, all the nodes are candidates (line 1). At each iteration λ, we pick A^λ, a random node among the candidates (line 3). Then, we select B^λ, one of the farthest candidates from A^λ (line 4) and C^λ, one of the farthest candidates from B^λ (line 5). B^λ and C^λ are extremities of the system composed with the candidates. Candidates for the $(\lambda + 1)^{th}$ iteration are the most equidistant nodes from B^λ and C^λ among the candidates at iteration λ (line 6). The most equidistant modules from B^λ and C^λ are defined as the modules that minimize $|distance\ to\ B^\lambda\ -\ distance\ to\ C^\lambda|$. This scheme stops when less than 3 nodes remain candidates. At this point, ABC-CenterV1 filters the set of candidates and only keeps the closest candidates from the last B and the last C (line 9). Then, ABC-Center picks one of the remaining candidate nodes as the approximate center of the network (line 10).

Ties are broken arbitrarily. In our distributed implementation, A^1 is the node of minimum identifier. Ties in the next node selections are broken using node identifiers in ABC-CenterV1 and at random in ABC-CenterV2.

The computation scheme of our two versions of ABC-Center differs in the selection of the central node when only two nodes remain candidate after the final step. ABC-CenterV2

picks a node at random, while ABC-CenterV1 selects one of the most equi-distant node on a shortest path from B to C (lines 8-9).

Figure 3.9 shows the ABC-CenterV2 step-by-step execution on a cube of 64 Blinky Blocks. Let $\mathscr{P}^\lambda$ be the plane that contains the most equidistant candidate blocks from B^λ and C^λ. The candidates at iteration $\lambda+1$ are the blocks that belong to $\bigcap_{\kappa=1..\lambda} \mathscr{P}^\kappa$. At each iteration, the problem is simplified by one dimension. The first iteration gives a discrete plane, the second a discrete line and the third a set of two blocks. One of the two remaining candidates is then arbitrarily selected as the center at the end of the third iteration.

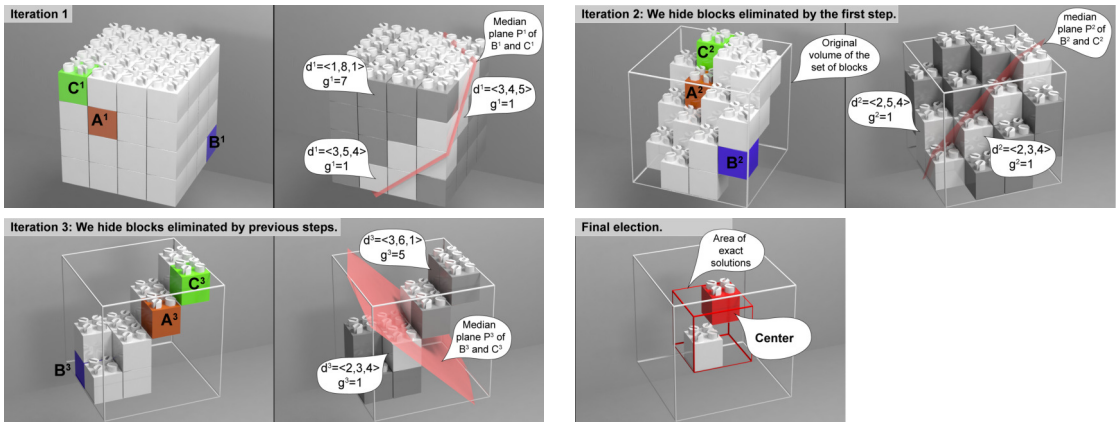


Figure 3.9: ABC-CenterV2 step-by-step execution on a $4\times4\times4$ cube of Blinky Blocks. For every block v_i we note $d^\lambda_{v_i} = <d(v_i,A^\lambda), d(v_i,B^\lambda), d(v_i,C^\lambda)>$ and $g^\lambda_{v_i} = |d(v_i,B^\lambda) - d(v_i,C^\lambda)|$.

Figure 3.10 shows that the position of the initial node, A^1, impacts the execution of ABC-Center in terms of both accuracy and efficiency (number of steps, i.e., time, number of messages, etc.). We recall that A^1 is the minimum identifier module, thus the execution of ABC-Center depends on the node identifier distribution.

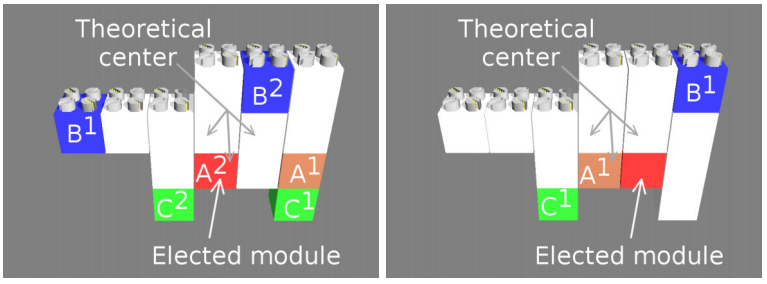


Figure 3.10: Two executions of ABC-CenterV2 on the same system with different positions for A^1. In the system on the left, the elected module belongs to the theoretical center, while it is one module off in the system on the right.

As experimentally shown in Section 3.9, ABC-Center exhibits a high accuracy in many systems. Nevertheless, we identified a tricky case (see Figure 3.11). In this example, all the modules in the right arm of Blinky Blocks are equidistant from B^1 and C^1 and

thus remain candidates for a second step, although they do not belong to the theoretical center. To solve the issue, we envisioned to keep only the modules on the shortest path between B^1 and C^1 as candidates for the second step, i.e., to replace line 6 by line 9 in Algorithm 7. As shown in Figure 3.11, this approach requires only one step and elects an exact center. However, we experimentally observed that using this method decreases the overall precision of our algorithm in large-scale compact random systems of Blinky Blocks. Hence, this very specific case remains to be investigated in future work.

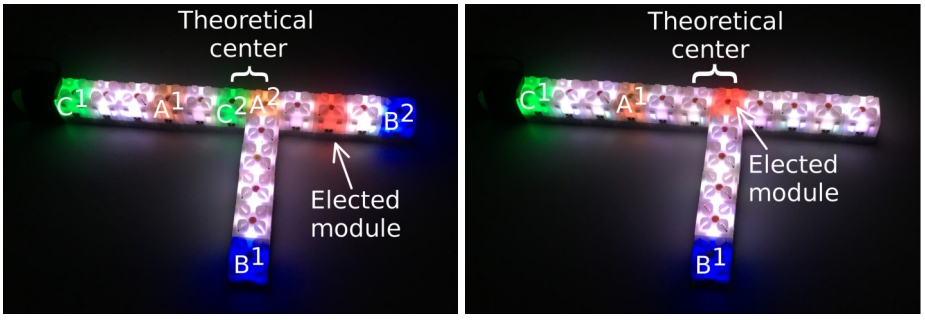


Figure 3.11: Specific ABC-Center approximation error case. On the left, execution with ABC-Center. On the right, execution with an approach we envisioned but abandoned.

3.7.2/ ABC-CenterV1: Distributed Implementation

Our distributed version of ABC-CenterV1, uses for each iteration λ, a multi-criterion leader election algorithm to find A^λ, B^λ and C^λ. We first describe this election algorithm. We subsequently detail step-by-step how the distributed version of ABC-CenterV1 works on a basic example. We then discuss the complexity of this version.

3.7.2.1/ Multi-Criterion Leader Election Algorithm

Our multi-criterion leader election algorithm is based on CHEUNG-BFS-ST algorithm. We recall that network traversal algorithms such as Cheung's algorithm can be used for leader election. Note that CHEUNG-BFS-ST does not use the controlled-broadcast optimization presented in Section 3.5.1.2.

We modified CHEUNG-BFS-ST into $electBlock(c, optFunc, x, id)$ to elect, among the candidate nodes for which the boolean c is equal to true, a single node that optimizes a variable x according to $optFunc \in \{min, max\}$ using the node identifier as a tie breaker (see Algorithm 8). Each node has its own variables c, x and id. c is equal to true if the module is candidate for the election, false otherwise. x can be a tuple. A comparison order has to be defined on x. In case of equality, the tuple with the lowest id is selected.

In $electBlock$, every node locally stores the temporary optimized value of x in the variable $optX$. The id of the candidate node that optimizes x is stored in $optId$ and the distance to that node is stored in $optDist$. The values of $optX$, $optId$ and $optDist$ are progressively learned by all the nodes during the execution of $electBlock$.

```
1  electBlock(c,optFunc,x,id) algorithm detailed for any node v_i:

2  Initialization of v_i:
3  optX ← WORST_X_VALUE; optId ← 0;
4  optDist ← 0; parent ←⊥; Wait ← ∅;
5  if c = true then
6      optX ← x; // value of the variable we want to optimize during the election
7      optId ← id;
8      for each v_j ∈ N^1_{v_i} do
9          send ELECT<optX, optId, optDist> to v_j;
10         wait ← wait ∪ {v_j};

11 When ELECT<x, i, d> is received by v_i from v_j do:
12 if (evaluation(optFunc,x,i) = BETTER) OR ((evaluation(optFunc,x,i) = EQUAL AND (optDist > d+1))
   then
13     if (evaluation(optFunc,x,i) = EQUAL) AND (parent != ⊥) then
14         send CONFIRM<optX, optId, optDist − 1> to parent;
15     optX ← x; optId ← i; optDist ← d + 1;
16     parent ← v_j; Wait ← ∅;
17     for each v_j ∈ N^1_{v_i} \ {v_j} do
18         send ELECT<optX, optId, optDist> to v_j;
19         Wait ← Wait ∪ {v_j};
20     if Wait = ∅ then
21         send CONFIRM<optX, optId, optDist − 1> to parent;
22 else if evaluation(optFunc,x,i) = EQUAL then
23     send CONFIRM<x, i, d> to v_j;

24 When CONFIRM<x, i, d> is received by v_i from v_j do:
25 if (evaluation(optFunc,x,i) = EQUAL) AND (optDist = d) then
26     Wait ← Wait − {v_j};
27     if Wait = ∅ then
28         if optId = id then
29             // Node v_i wins the election
30         else
31             send CONFIRM<optX, optId, optDist − 1> to parent;
```

Algorithm 8: Multi-criterion leader election algorithm $electBlock(c, optFunc, x, id)$ detailed for any node v_i.

The evaluation function $evaluation(optFunc, x, i)$ returns $BETTER$ if the tuple (x, i) optimizes the local solution $(optX, optId)$ according to $optFunc$. It returns $EQUAL$ if $(x, i) = (optX, optId)$. Otherwise, it returns $WORSE$. For instance, if $optX = 2$ and $optId = 1$, $evaluation(max, 3, 2)$ returns $BETTER$. The same call, returns $EQUAL$ if $optX = 3$ and $optId = 2$, whereas it returns $WORSE$ if $optX = 3$ and $optId = 1$.

In our leader election algorithm, each candidate node starts a network traversal by sending to all its neighbors an $ELECT$ message that contains its value of x and its id (lines 2-10). Network traversals are concurrent. If a node receives better values according to $optFunc$ via an $ELECT$ message, it forgets about the previous network traversal, and starts participating in the new one (lines 11-23). Modules send back confirmation messages $CONFIRM$ which progressively go back up to the module that will win the election. A module v_i sends a $CONFIRM$ message to the node v_j, either if v_i has received from v_j

an $ELECT$ message containing values equal to the current optimal values stored in $optX$ and $optId$ but with a farther distance to the node of identifier $optId$ (lines 14 and 23), or if v_i has received a $CONFIRM$ message from each of its other neighbors (lines 21 and 31). A graph traversal terminates as soon as the module that initiated it, has been informed by all its neighbors that it has the best values for x and id among the candidate nodes (line 29). Although all modules initiate a network traversal, only a single one will terminate and the node that initiates this traversal will win the election.

3.7.2.2/ ABC-CenterV1 Detailed Execution on a Line of 4 Blinky Blocks

Figure 3.12 shows ABC-CenterV1 step-by-step execution on a line of four blocks. Election messages are tagged with the iteration number and the role (A,B or C) to prevent the current election from being disrupted by delayed messages of a previous election. At the beginning every block is candidate and launches the election of A^1. When A^1 is finally elected, all the blocks know their distance to A^1. A^1 starts the election of B^1 which is one of the farthest blocks from A^1. Similarly, B^1 starts the election of C^1. Then, C^1 launches the election of A^2. Only the blocks k that have $g_k^1[0] = g_{A^2}^1[0]$ remain candidates for the second iteration, i.e., the blocks at half-distance between B^1 and C^1. If less than 3 blocks remain candidates, A^2 is elected as an approximate center of the system and the algorithm terminates. Otherwise, the same scheme is repeated until less than 3 blocks remain candidates. An easy way to determine if less than 3 blocks remain candidates is to store the identifiers of 2 remaining candidates in the $CONFIRM$ election messages. If A^λ receives 2 different identifiers, it means that at least 3 blocks remain candidates: A^λ and the other two. The message size is thus constant.

3.7.2.3/ Termination Proof and Complexity Analysis

M. Raynal's termination proof for Cheung's BFST algorithm [Raynal, 2013] is applicable to show the termination of our multi-criterion leader election. Since the number of candidate nodes always decreases at each iteration, the ABC-CenterV1 algorithm necessarily terminates.

Cheung's algorithm has a message complexity of $O(nm)$, where m is the number of edges [Lynch, 1996]. During the multi-criterion leader election process, in the worst case, each node launches a graph traversal and a single one terminates. Three elections are performed at each step. Hence, the message complexity of ABC-CenterV1 is $O(smn^2)$ with s representing the number of steps.

If we ignore pileups, Cheung's BFST algorithm has a time complexity of $O(d)$ [Raynal, 2013], as a node is at most at d hops from any other node. Under the same assumption, our multi-criterion leader election algorithm also has a time complexity of $O(d)$. A step is composed of exactly three multi-criteria elections, thus, the time complexity of a step is $O(d)$. Hence, ABC-CenterV1 uses $O(sd)$ time with s denoting the number of iterations required for termination. However, ABC-CenterV1 does not use the controlled-broadcast optimization presented in Section 3.5.1.2 and, thus, messages may pile up, incurring a

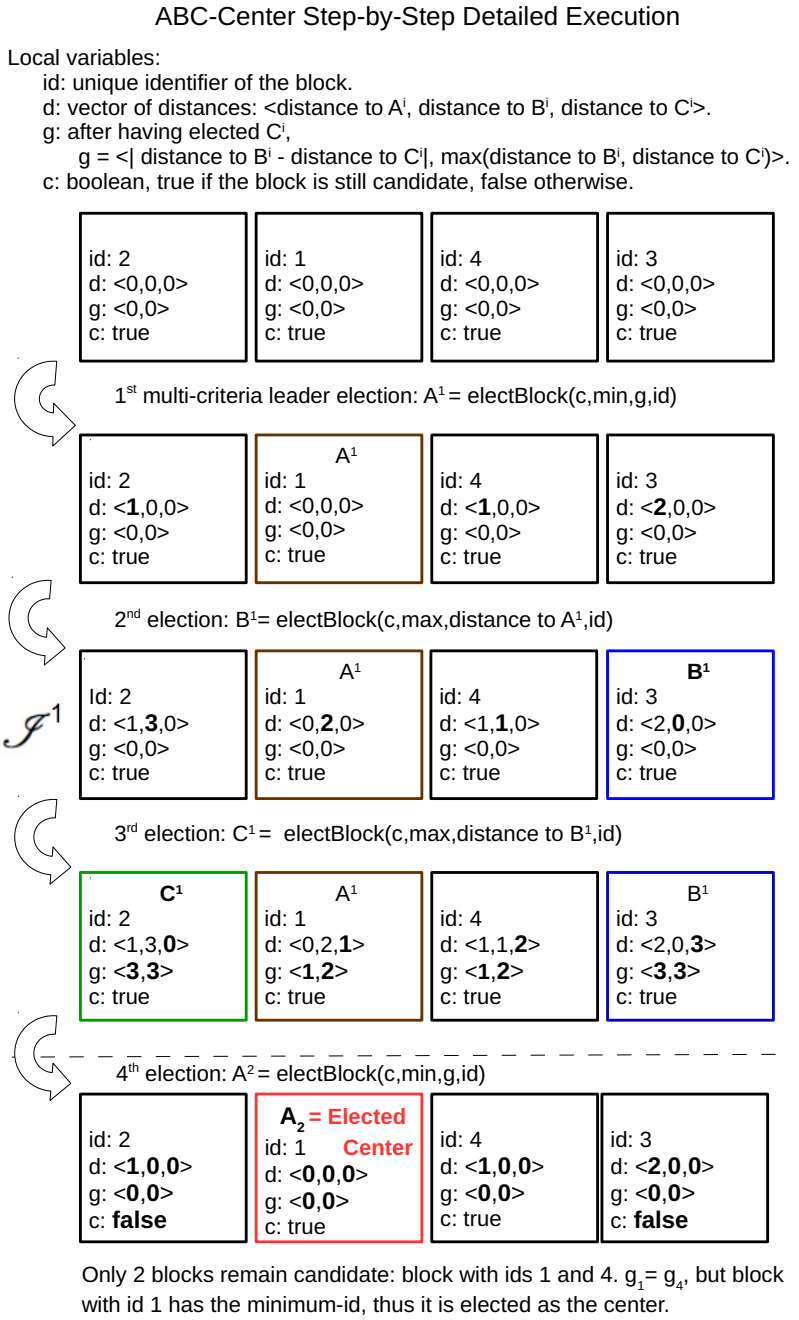


Figure 3.12: ABC-CenterV1 detailed execution on a line of 4 Blinky Blocks.

time overhead.

The ABC-CenterV1 algorithm stores $O(\Delta)$ information at the application level. However, its total memory usage can be important, due to potential message pileups.

ABC-CenterV2 presented in the next section overcomes these limitations of ABC-CenterV1.

3.7.3/ ABC-CENTERV2: DISTRIBUTED IMPLEMENTATION

3.7.3.1/ PSEUDO-CODE

Our distributed implementation of ABC-CenterV2 is described in Algorithm 9. It uses three primitives (LE_CHEUNG-BFS-ST-CB_STB-STC, CHEUNG-BFS-ST-CB-AGG and STB-STC) and two specific types of messages (NEXT_BFS and ELECTED).

A^1 is elected using LE_CHEUNG-BFS-ST-CB_STB-STC (lines 5 and 49). If there are more than 2 nodes in the system, a NEXT_BFS message is sent toward B^1, a farthest candidate node from A^1 (lines 50-51 and 66-81). Otherwise, A^1 is elected as the cen-

Output : a single central node is elected
Primitive(s) : LE_CHEUNG-BFS-ST-CB_STB-STC
CHEUNG-BFS-ST-CB-AGG($handlers$: $handleBFSData, updateBFSAggs, getBFSAggs, resetBFSAggs$)
STB-STC($tree$: CHEUNG-BFS-ST-CB-AGG.$tree$, $handlers$: $stbHandler, stcHandler$)

```
   // Initialization and start handlers:
1  Initialization of v_i:
2  candidate ← true; iteration ← 0; distances ← <0,0>;
     branchFarthest ← {}; farthest ← 0;
     nextHopToFarthest ←⊥; branchGMin ← {};
     gMin ← 0; nextHopToGMin ←⊥; numCandidates ← 0;
3  start ABC-CenterV2;

4  When ABC-CenterV2 starts at node v_i do:
5  start LE_CHEUNG-BFS-ST-CB_STB-STC;

   // Helper functions:
6  Function updateBCDistances():
7      role ← iteration%3;
8      if role > 0 then // is B or C
9          distances[role − 1] ←
10             CHEUNG-BFS-ST-CB-AGG.distance;

   // Primitive handlers for aggregate computation
      and data propagation:
11 Function handleBFSData():
12     iter ← CHEUNG-BFS-ST-CB-AGG.data[0];
13     if iter > iteration then
14         iteration ← iter;
           // Take part in this BFS as non-root:
15         re-initialize CHEUNG-BFS-ST-CB-AGG;

16 Function updateBFSAggs(v_j, child, aggs) :
17     if child = true then
18         branchFarthest[v_j] = aggs[1];
19         branchGMin[v_j] = aggs[2];
20     else
21         remove branchFarthest[v_j], branchGMin[v_j];

22 Function resetBFSAggs():
23     branchFarthest ← {}; branchGMin ← {};

24 Function getBFSAggs:
25     updateBCDistances();
26     if candidate = true then
27         farthest ← CHEUNG-BFS-ST-CB-AGG.distance;
28         gMin ← |distances[0] − distances[1]|;
29     else
30         farthest ← 0; gMin ← +∞;
31     nextHopToFarthest ←⊥;
32     v_f ← argmax_{v_k∈N^1_{v_i}} branchFarthest[v_k];
33     if v_f ≠⊥ AND branchFarthest[v_f] > farthest then
34         farthest ← branchFarthest[v_f];
           nextHopToFarthest ← v_f;
35     nextHopToGMin ←⊥;
       v_g ← argmin_{v_k∈N^1_{v_i}} branchGMin[v_k];
36     if v_g ≠⊥ AND branchGMin[v_g] < g then
37         g ← branchGMin[v_g]; nextHopToGMin ← v_g;
38     CHEUNG-BFS-ST-CB-AGG.aggregates ←
       CHEUNG-BFS-ST-CB-AGG.aggregates ∪
       {farthest, gMin};

39 Function handleSTB() :
40     nextHopToGMin ←⊥; numCandidates ← 1;
       g ← |distances[0] − distances[1]|;
41     if g > STB-STC.data[0] then
42         candidate ← false; numCandidates ← 0;
43     STB-STC.aggregates ← {numCandidates};

44 Function handleSTC(v_j, aggs) :
45     numCandidates ← numCandidates + aggs[0];
46     if nextHopToGMin =⊥ AND aggs[0] ≠ 0 then
47         nextHopToGMin ← v_j;
48     STB-STC.aggregates ← {numCandidates};
```

```
   // Primitive termination handlers:
49 When LE_CHEUNG-BFS-ST-CB_STB-STC terminates
   at root node v_i do:
50 if LE_CHEUNG-BFS-ST-CB_STB-STC.size > 2 then
51 |  send NEXT_BFS<LE_CHEUNG-BFS-ST-CB_STB-
   |  STC.size> to
   |  LE_CHEUNG-BFS-ST-CB_STB-
   |  STC.nextHopToFarthest;
52 else
53 |  ABC-CenterV2 terminates; // v_i is elected

54 When CHEUNG-BFS-ST-CB-AGG terminates at root
   node v_i do:
55 updateBCDistances();
56 if iteration%3 ≠ 2 then // is A or B
57 |  send NEXT_BFS<CHEUNG-BFS-ST-CB-AGG.size>
   |  to nextHopToFarthest;
58 else
59 |  STB-STC.data ← {gMin};
60 |  start STB-STC;

61 When STB-STC terminates at root node v_j do:
62 if numCandidates > 2 then
63 |  send NEXT_BFS<CHEUNG-BFS-ST-CB-AGG.size>
   |  to STB-STC.nextHopToGMin;
64 else
65 |  send ELECTED<> to STB-STC.nextHopToGMin;

   // ABC-CenterV2 message handlers:
66 When NEXT_BFS<size> message is received by the
   node v_i do:
67 pathNextBFS ← nextHopToFarthest;
68 if iteration = 0 then // is A^1
69 |  pathNextBFS ← LE_CHEUNG-BFS-ST-CB_STB-
   |  STC.nextHopToFarthest;
70 if iteration%3 = 2 then // is C
71 |  pathNextBFS ← STB-STC.nextHopToGMin;
72 if pathNextBFS =⊥ then
73 |  iteration ← iteration + 1;
74 |  if iteration%3 ≠ 0 then // is B or C
75 |  |  candidate ← false;
   |  // Start a new BFS as root:
76 |  re-initialize CHEUNG-BFS-ST-CB-AGG;
77 |  CHEUNG-BFS-ST-CB-AGG.size ← size;
78 |  CHEUNG-BFS-ST-CB-AGG.data[0] ← iteration;
79 |  start CHEUNG-BFS-ST-CB-AGG;
80 else
81 |  send NEXT_BFS<size> to pathNextBFS;

82 When ELECTED<> message is received by node v_i do:
83 if candidate then
84 |  ABC-CenterV2 terminates; // v_i is elected
85 else
86 |  send ELECTED<> to STB-STC.nextHopToGMin;
```

Algorithm 9: ABC-CenterV2 detailed for any node v_i.

tral node and ABC-CenterV2 terminates (line 53). Upon reception of that NEXT_BFS message, B^1 initiates a CHEUNG-BFS-ST-CB-AGG (lines 72-79) during which all nodes determine their distance to B^1 and a path from B^1 toward C^1, a farthest candidate node from B^1, is constructed (lines 16-38). Similarly to A^1, B^1 sends a NEXT_BFS message to C^1 that initiates a CHEUNG-BFS-ST-CB-AGG during which modules distributively compute $gMin^1 = \min_{v_j \in Candidates} g^1_{v_j} = \min_{v_j \in Candidates} |d(v_j, B^1) - d(v_j, C^1)|$ (lines 16-38).

Afterwards, C^1 initiates an STB-STC (line 60). During the STB phase, $gMin^1$ is broadcast across the network (line 59). Only the candidate nodes v_j with $g^1_{v_j} = gMin^1$ remain candidates for the second step (lines 39-43). The STC phase is used to determine the number of remaining candidate nodes and to construct a path toward a candidate node (lines 44-48). If less than 3 nodes remain candidates, C^1 sends an ELECTED message toward one of these candidate nodes (lines 65 and 82-86). ABC-CenterV2 terminates upon reception of an ELECTED message by a candidate node which is elected as the central node (line 84). Otherwise, if more than 2 nodes remain candidates, C^1 sends a NEXT_BFS message to A^2, one of the remaining candidate nodes (lines 63 and 66-81). A^2 initiates a CHEUNG-BFS-ST-CB-AGG to locate B^2, the farthest candidate node from A^2 (lines 72-79). At this point, the execution of the second step is identical to the execution of the first one after the NEXT_BFS message has reached B^1. The scheme of the

second iteration is repeated at every step until less than 3 nodes remain candidates.

3.7.3.2/ TERMINATION PROOF AND COMPLEXITY ANALYSIS

Let s be the number of iterations required by ABC-CenterV2 to terminate. Our algorithm first runs a LE_CHEUNG-BFS-ST-CB_STB-STC, then $(3s-1)\times$ CHEUNG-BFS-ST-CB-AGG and $s\times$ STB-STC in a sequential way. All these primitives have been proved to terminate. Since the number of candidate nodes always decreases at each iteration, the ABC-CenterV2 algorithm necessarily terminates. Moreover, at least two nodes (i.e., B and C) are eliminated at each step, thus $s \leq n$.

The NEXT_BFS and ELECTED messages use $O(d)$ time and $O(n)$ messages to reach their final destination. Hence, using the primitive complexity given in Section 3.5, ABC-CenterV2 runs in $O(sd)$ time using $O(mn^2)$ messages and $O(\Delta)$ memory space per module.

3.8/ PROBABILISTIC-COUNTER-BASED CENTRAL-LEADER ELECTION FRAMEWORK

This section presents the Probabilistic-Counter-based Central-Leader Election Framework (PC2LE) designed to elect either an approximate center node or an approximate centroid node. PC2LE is an extended version of our algorithm presented in [Naz et al., 2016] and combines the idea introduced in the input-graph analysis algorithms [Kang et al., 2011a, Kang et al., 2011b] and in the distributed synchronous algorithm [Garin et al., 2012].

3.8.1/ PROBABILISTIC COUNTERS

PC2LE is based on probabilistic counting. Probabilistic counters are designed to estimate the number of unique elements in a set, using both a low time complexity and a low memory footprint.

Any probabilistic counter can be used (e.g., the Flajolet-Martin [Flajolet et al., 1985], the HyperLogLog [Flajolet et al., 2007] and the counters proposed in [Varagnolo et al., 2010]). Note that the choice of the counter has an impact on the precision of PC2LE and on its memory requirements. As explained in the evaluation section, we obtained the most accurate results using the HyperLogLog counter.

A probabilistic counter comes with 3 operations, namely the initialization, merge and estimate size operations. The initialization operation initializes the probabilistic counter and encodes a single initial value in its set. The merge operation makes it possible to merge two probabilistic counters. The size of the set encoded by a probabilistic counter is estimated using the estimate size function.

Flajolet-Martin uses h bitstrings of $\log_2 w$ bits each to estimate the number of distinct el-

ements, L, with $L \leq w$, in a set of items with a standard error of $O(\frac{0.78}{\sqrt{h}})$ [Flajolet et al., 1985]. In turn, HyperLogLog uses k registers of $O(\log\log w)$ bits each to provide a standard error of $\frac{1.04}{\sqrt{k}}$ where k is the number of registers [Flajolet et al., 2007]. h and k are input parameters. Thus, the actual memory usage is a design choice. In practice, there is a trade-off between the memory requirement of a counter, its precision and the number of elements to be counted.

These counters have a low work complexity. In Flajolet-Martin, the add, merge and count functions require $O(h)$ operations [Gibbons, 2016]. In HyperLogLog, the add, merge and count functions respectively need $O(1)$, $O(k)$ and $O(k)$ operations. Note that we assume the call to the hash function performed by the add function to be a constant time operation.

3.8.2/ DESCRIPTION AT A GLANCE

For pedagogical purposes, a sequential version of the PC2LE framework is shown in Algorithm 10.

Input : $G = (V, E)$ // network representation
$version \in \{center, centroid\}$
Output : $centrality[]$ // approximate eccentricity or farness of every node
$central$ // approximate center or centroid of the system

1 **for each** $v_i \in V$ **do**
2 $PC[v_i][0] \leftarrow init(id_{v_i})$;// PC is an $n \times 2$ matrix of probabilistic counters that encodes the sets of reachable nodes at the current/previous iteration for every "line" node.
3 $centrality[v_i] \leftarrow 0$;
4 $I \leftarrow v_i \in \underset{v_j \in V}{\operatorname{argmin}}\, id_{v_j}$; // node of minimum-identifier
5 $\bar{d} \leftarrow 2 \times \max_{v_i \in V} d(v_i, I)$// Compute an upper-bound on the network diameter
6 **for** r = *1 to* $\bar{d}$ **do**
7 $prev \leftarrow (r-1)\%2$;
8 $cur \leftarrow r\%2$;
9 **for each** $v_i \in V$ **do**
10 **for each** $v_j \in N^1_{v_i}$ **do**
11 $PC[v_i][cur] \leftarrow merge(PC[v_i][prev], PC[v_j][prev])$;
12 **if** $version = centroid$ **then**
13 $sizeCur \leftarrow estimateSize(PC[v_i][cur])$; // $\approx |N_{v_i}(r)|$
14 $sizePrev \leftarrow estimateSize(PC[v_i][prev])$; // $\approx |N_{v_i}(r-1)|$
/* Farness estimation using Equation (3.6): */
15 $centrality[v_i] \leftarrow centrality[v_i] + r * (sizeCur - sizePrev)$;
16 **else** // center
17 **if** $PC[v_i][cur] \neq PC[v_i][prev]$ **then**
18 $centrality[v_i] = r$; // Eccentricity estimation using Equation (3.2)
19 $central \leftarrow v_i \in \underset{v_j \in V}{\operatorname{argmin}}\, centrality[v_j]$; // a node of minimum estimated centrality is elected

Algorithm 10: Sequential version of the PC2LE framework.

Every node v_i starts with a probabilistic counter $PC[v_i][0]$ encoding a set that contains only v_i itself and a null centrality value (lines 1-3). PC2LE runs in $O(d)$ rounds. At the end of

round r, the probabilistic counter of a node represents the set of nodes within r-hops from that node. At each round r, every node v_i updates its probabilistic counter by merging it with the $(r-1)$-round probabilistic counter of all its immediate neighbor nodes v_j (line 11). The centrality of every node is computed using Equations (3.2) or (3.6), depending on the version of the framework that is run (lines 12-18).

PC2LE requires d rounds to converge. The sequential algorithms presented in [Kang et al., 2011a, Kang et al., 2011b] continue their execution until there is no more update, i.e., the internal state of no probabilistic counter has changed. This method will be expensive in distributed settings as it requires all nodes to be queried at the end of every round. In [Garin et al., 2012], an upper bound of the diameter is assumed to be known or pre-computed using an external algorithm. PC2LE initially finds I, the minimum-identifier node and computes $\tilde{d} = 2ecc(I)$ as upper bound of the network diameter (lines 4-5). Indeed, the eccentricity of any given node v_i provides bounds of the diameter of the system: $ecc(v_i) \leq d \leq 2ecc(v_i) \leq 2d$ [Magnien et al., 2009]. Note that this implies that PC2LE runs in $O(d)$ rounds.

After $\tilde{d}$ rounds, a node of minimum centrality value is elected (line 19). PC2LE is approximately equivalent to running a BFSes from every node but at less memory and computation expense. Notice that the computed values depend on the probabilistic counter internal algorithms and on the node identifier distribution.

3.8.3/ Distributed Implementation

Algorithm 11 details the pseudo-code of the PC2LE framework for any node. Our framework uses three primitives (LE_CHEUNG-BFS-ST-CB_STB-STC, STB and STC) and two specific types of messages (UPDATE and ELECTED). PC2LE is composed of three steps.

During the first step, nodes run the LE_CHEUNG-BFS-ST-CB_STB-STC algorithm to elect an initiator, construct a spanning-tree and compute $\tilde{d}$, an upper bound of the network diameter (lines 9 and 28). $\tilde{d}$ is used to bound the execution of the second step. If there are 2 nodes or less in the system, the initiator is elected as the central node and PC2LE terminates (line 26).

Otherwise, the initiator then initiates an STB to broadcast $\tilde{d}$ across the network and to start the second-step computations (lines 28-30). During the second step, nodes compute their estimation of the node farness or eccentricity, depending on the running version of the framework, in $\tilde{d}$ synchronous rounds. Note that the second step actually embeds the Alpha synchronizer [Awerbuch, 1985, Lynch, 1996, Raynal, 2013] in it. At each round r, immediate neighbor nodes exchange their current probabilistic counter that encodes the set of nodes at distance $r-1$ to allow them to compute their next-round probabilistic counter (lines 37-58).

Upon termination of the second step, all nodes have an estimation of their centrality value (line 53). Finally, in the third step, nodes execute an STC over the tree constructed in the first step in order to elect the node of minimum estimated centrality value (line 55). If

the initiator elected in the first step has the minimum centrality value, PC2LE terminates upon the termination of STC and the initiator is elected as the central node (line 34). Otherwise, an ELECTED message is sent toward the node of minimum centrality value (lines 36 and 59-63). Upon reception of that message by the node of minimum centrality, PC2LE terminates and this node is elected as the central node (line 61).

3.8.4/ Termination Proof and Complexity Analysis

PC2LE uses the LE_CHEUNG-BFS-ST-CB_STB-STC and STC primitives. We recall that their complexity is given in Section 3.5. Let c denote the memory complexity of the probabilistic counter that is used in our framework.

PC2LE first elects an initiator using LE_CHEUNG-BFS-ST-CB_STB-STC. PC2LE then broadcasts a START message through the constructed tree to trigger the round-based centrality computation, using $O(d)$ time and $O(n)$ messages. Nodes then estimate their centrality values in $O(d)$ rounds as the estimation of the diameter is bounded by $2d$. During each round, every node exchanges two messages with all of its immediate neighbors.

Input : $version \in \{center, centroid\}$
Output : a single central node is elected
$centrality$ // v_i's approximate eccentricity or farness
Primitive(s) : LE_CHEUNG-BFS-ST-CB_STB-STC
STB($tree$: LE_CHEUNG-BFS-ST-CB_STB-STC.$tree$, $handler$: $startHandler$)
STC($tree$: LE_CHEUNG-BFS-ST-CB_STB-STC.$tree$, $handler$: $electionHandler$)

```
   // Initialization and start handlers:
1  Initialization of v_i:
2  round ← 0; bound ← 0 centrality ← 0;
3  pc ← init(id_{v_i});// probabilistic counter
4  npc^prev ← pc; npc^cur ← pc;// neighborhood
       probabilistic counters
5  received^prev ← 0; received^cur ← 0;
6  minCentrality ← +∞; nextHopToMinCentrality ←⊥;
7  start PC2LE;

8  When PC2LE starts at node v_i do:
9  start LE_CHEUNG-BFS-ST-CB_STB-STC;

   // Primitive handlers for aggregate computation
       and data propagation:
10 Function initNextRound():
11     for each v_k ∈ N^1_{v_i} do
12         send UPDATE(pc, round) to v_k;
13     round ← round + 1; npc^prev ← merge(npc^prev, npc^cur);
14     received^prev ← received^cur; received^cur ← 0;
15 Function startHandler():
16     bound ← STB.data;
17     initialize STC;
18     STC.waiting ← STC.waiting + 1;
19     initNextRound();
20 Function electionHandler(v_j, aggs):
21     if minCentrality > aggs[0] then
22         minCentrality ← aggs[0];
23         nextHopToMinCentrality ← v_j;
   // Primitive termination handlers:
24 When LE_CHEUNG-BFS-ST-CB_STB-STC terminates at
       root node v_i do:
25 if LE_CHEUNG-BFS-ST-CB_STB-STC.size < 3 then
26     PC2LE terminates; // v_i is elected
27 else
28     bound ←
         2 × LE_CHEUNG-BFS-ST-CB_STB-STC.height;
29     STB.data ← bound;
30     start STB;
31     startHandler();

32 When STC terminates at root node v_i do:
33 if nextHopToMinCentrality =⊥ then
34     PC2LE terminates; // v_i is elected
35 else
36     send ELECTED<> to nextHopToMinCentrality;
```

```
   // PC2LE message handlers:
37 When UPDATE(c, r) is received by v_i do:
38 if round < bound then
39     if round = r + 1 then
40         received^prev ← received^prev + 1;
41         merge(npc^prev, c);
42         if received^prev = |N^1_{v_i}| then
43             if version = center then
44                 if pc ≠ npc^prev then
45                     centrality ← round;
46                 merge(pc, npc^prev);
47             else
48                 size^prev = estimateSize(pc);
49                 pc ← merge(pc, npc^prev);
50                 size^cur = estimateSize(pc);
51                 centrality ←
                     centrality + round ∗ (size^cur − size^prev);
52             initNextRound();
53             if round = bound − 1 then
                   // v_i starts the minimum
                      centrality value election.
                   electionHandler(⊥, centrality);
54                 STC.waiting ← STC.waiting − 1;
55                 start STC;
56     else // r = round
57         received^cur ← received^cur + 1;
58         npc^cur ← merge(npc^cur, c);
59 When ELECTED<> message is received by node v_i do:
60 if nextHopToMinCentrality =⊥ then
61     PC2LE terminates; // v_i is elected
62 else
63     send ELECTED<> to nextHopToMinCentrality;
```

Algorithm 11: Distributed implementation of the PC2LE framework detailed for any node v_i.

Thus, nodes exchange at most $O(m)$ messages of size $O(c)$ per round.

Afterwards, nodes run an STC toward the initiator which then sends an ELECTED message toward the node of minimum centrality value. The elected node is, at most, at distance d from the initiator, thus at most $O(d)$ time and messages are required to route a message from the initiator to the elected node through the tree rooted at the initiator.

All the steps described above terminate, thus, PC2LE terminates. Moreover, our framework converges in $O(d)$ time using $O(\Delta+c)$ memory space per node and $O(mn^2)$ messages of size $O(c)$.

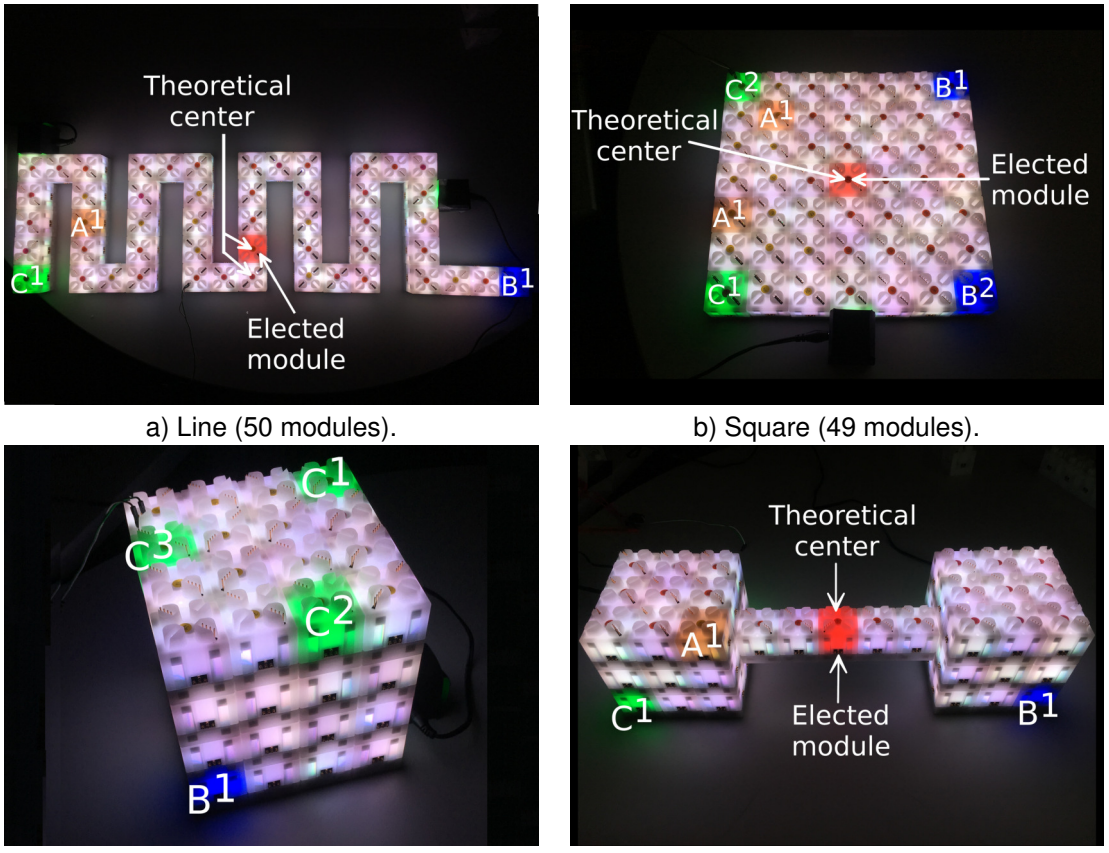
3.9/ EVALUATION

This section presents our experimental evaluation performed both on hardware Blinky Blocks and in the VisibleSim simulator (see Sections 2.3.1 and 2.3.3). Through our experiments, we show the effectiveness, the efficiency and the scalability of our algorithms. More precisely, we first show that ABC-CenterV1 works well on hardware through some

examples. Then, we present our simulation model and show that VisibleSim accurately simulates the Blinky Blocks behavior. Finally, we use VisibleSim to evaluate the performance of our algorithms in large-scale systems and to compare them to existing algorithms in terms of accuracy, execution time, number of messages and memory usage.

3.9.1/ Evaluation of ABC-CenterV1 on Hardware

We implemented ABC-CenterV1 in C and evaluated on the Blinky Blocks hardware. Figure 3.13 shows ABC-CenterV1 results in some basic configurations with hardware Blinky Blocks. For all the configurations considered, the computed center exactly match one of the nodes in the exact center of the systems. For your information, ABC-CenterV2 successfully finds an exact-center node in these configurations as well.



a) Line (50 modules). b) Square (49 modules).

c) Cube (64 modules). d) Dumbbell (59 modules).

Figure 3.13: ABC-CenterV1 executions on different hardware Blinky Blocks configurations.

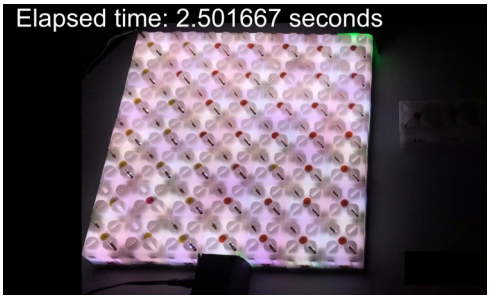
Table 3.4 gives the execution times of ABC-CenterV1 in these configurations on different scales formed from 5 to 64 modules. The execution time depends on the diameter of the system and on the number of steps of our algorithm. We observe that the number of steps does not only depend on the geometrical dimension of the system shape (e.g., a 3D cube needs 3 steps, while a 3D dumbbell requires only 1 step like the line).

Figure 3.14 illustrates ABC-CenterV1 tolerance to network dynamics. The system initially

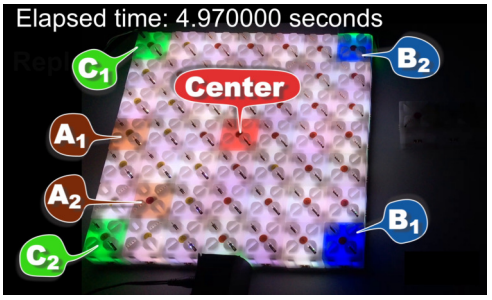
Shape	Size (module)	Diameter (hop)	ABC-CenterV1		
			# steps	Average execution time ± standard deviation (ms)	
				Hardware	Simulator
Line	5	4	1	234 ± 1	244 ± 3
	10	9		545 ± 5	544 ± 5
	50	49		2873 ± 23	2885 ± 17
Square	9	4	2	598 ± 45	588 ± 14
	25	8		1117 ± 30	1119 ± 27
	49	12		1684 ± 48	1686 ± 44
Cube	27	6	3	1229 ± 56	1214 ± 31
	64	9		1927 ± 51	1941 ± 33
Dumbbell	59	15	1	1262 ± 56	1252 ± 57

Table 3.4: Average execution time of ABC-CenterV1 on hardware Blinky Blocks and in simulations. Statistics on the execution time were computed over 25 runs for every configuration. Simulation timing results were computed several times, each time on 25 independent runs, and we kept the values that matched best the hardware execution time.

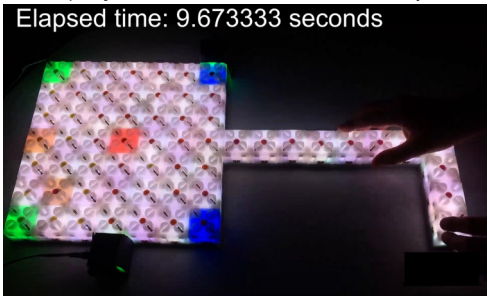
forms a 7x7 square. Modules take about 2.5 seconds to boot up, initialize themselves, discover their neighborhood and elect their center. An extra arm of 11 Blinky Blocks is then connected to the square-shaped ensemble which detects the network change and elects its new center in approximately 2 seconds. Note that ABC-CenterV1 is locally launched on a module at the earliest: 1 second after the module complete initialization using a software timeout, 1 second after a neighbor change detection or upon reception of an ABC-CenterV1 message. New neighbors are detected in a few hundred milliseconds.



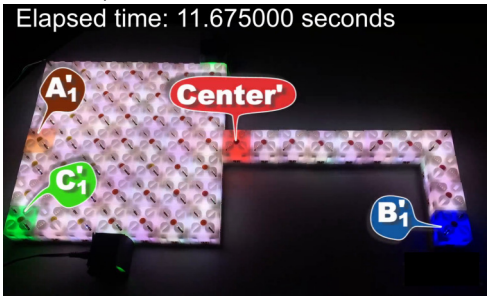
a) System of 49 nodes starts up.



b) ABC-Center terminates.



c) Adding 11 nodes.



d) ABC-Center terminates.

Figure 3.14: ABC-CenterV1 execution in a dynamic network.

3.9.2/ Simulation Model and Fidelity

This section shows the simulation model we have implemented in VisibleSim to simulate the behavior of the Blinky Blocks. Our model takes into account the processing time, the queuing time and the communication time (see Table 3.5).

Parameters		Value
Transfer rate $(kbit/s)$	$\lvert N^1_{v_i}\rvert \leq 2$	$\mathcal{N}(28.331, 1.112)$
	$2 < \lvert N^1_{v_i}\rvert \leq 4$	$\mathcal{N}(26.667, 2.471)$
	$4 < \lvert N^1_{v_i}\rvert \leq 6$	$\mathcal{N}(25.846, 2.471)$
Processing time (s)		$\mathcal{U}(25 \times 10^{-6}, 125 \times 10^{-6})$

Table 3.5: Communication model. $\mathcal{N}(\mu, \sigma)$ refers to the normal probabilistic law with μ being the mean and σ the standard deviation. $\mathcal{U}(l, u)$ refers to the uniform probabilistic law with the minimum value l and the maximum value u.

The processing time represents the time necessary to handle an incoming message. We used the micro-controller clock running at 32 MHz (nano-second scale resolution) to measure the processing time of different message handlers and arbitrarily chose to simulate the processing time using a uniform distribution with the range of the measured time.

In our work on time synchronization presented in Chapter 4, we estimate the transfer rate between neighboring modules using round-trip time measurements (see Section 4.4). The transfer rate corresponds to the communication rate from the data-link layer to the data-link layer of neighboring nodes. We observed that the transfer rate depends on the number of simultaneous communications. In this section model, the communication rate depends on the size of the neighborhood of a node.

The reader may have noted that the simulation model presented here differs from the model we use in Chapter 4. There are several reasons for that. Firstly, we use here some newly fabricated Blinky Blocks hardware with some different hardware components (e.g., the network connectors). Secondly, their firmware is slightly modified as well. In particular, we have reduced the time a module needs to find a free frame using dynamic frame allocation instead of a static array of frames with a free-frame search cost of $O(\#\text{ static frames})$. This reduces the message processing time as modules require less time to send messages in response to incoming ones. Last but not least, we use here a less fine-grained simulation model for the sake of time efficiency. For instance, we do not check every single byte of a message for special bytes to escape; instead, we only increase the average and the standard deviation of the communication rate to mimic that phenomenon. We slightly adapt the transfer rate in order to obtain simulation times that match the ABC-CenterV1 execution time on new Blinky Blocks hardware prototypes.

Table 3.4 shows that the simulated execution times on VisibleSim closely match the execution time obtained experimentally on hardware Blinky Blocks, for small and larger con-

figurations, and for sparse (e.g., lines), less-sparse (e.g., squares), compact (e.g., cubes) and mixed-density configurations with compact components linked by a critical path (e.g., the dumbbell). Thus, VisibleSim can be used to accurately benchmark the performance of our algorithms on much bigger configurations.

3.9.3/ Large-scale Evaluation and Comparison to Existing Algorithms

We use VisibleSim to evaluate the performance of our algorithms and to compare them with existing ones in terms accuracy, execution time, number of messages and memory usage on random large-scale Blinky Block systems. Random systems were generated by connecting the modules one by one to the system at random, starting from a single node. This guarantees the connectivity of the network and tends to generate compact systems with a reasonable diameter. Modules have a unique identifier in $\{1..n\}$. Unless explicitly mentioned, every single point on the result plots represents 50 independent executions.

3.9.3.1/ Compared Algorithms and Parameters

We compare our algorithms to several approaches that we potentially ported to fit our system models.

Our work: We consider ABC-CenterV1, ABC-CenterV2, k-BFS SumSweep, PC2LE and the algorithm we proposed in [Naz et al., 2016]. We use the following parameters:

- In the k-BFS SumSweep framework, we arbitrarily choose $k = 10$.
- In our implementation of PC2LE, we use the HyperLogLog [Flajolet et al., 2007] probabilistic counter using 16 registers of 5 bits each for a total of 80 bytes with the 32-bit Knuth multiplicative hash function [Knuth, 1998]. Actually, we experimentally compared several combinations of counters (the Flajolet-Martin [Flajolet et al., 1985] and HyperLogLog [Flajolet et al., 2007] counters) and hash functions (affine functions, Knuth's multiplicative hash functions [Knuth, 1998], the MurMur3 [Appleby, 2011] hash function and the FNV hash function [Fowler et al., 1991]). Probabilistic counting involves a trade-off between the memory space used by the counter and its accuracy. In our tests, we limit the size of the different counters so that any PC2LE message fits into a single Blinky Blocks frame, i.e., a counter can occupy 10 bytes at most. We choose the HyperLogLog along with the Knuth multiplicative hash function as it leads to more accurate results. For the reader's information, the Flajolet-Martin counter based on five 16-bit affine functions $h(x) = ax + b$, where a and b are small odd numbers, also performs very well.
- In [Naz et al., 2016], we proposed the E2ACE (Efficient and Effective Approximate-Centroid Election) algorithm which approximately corresponds to the centroid version of PC2LE based on the Flajolet-Martin probabilistic counter combined with the identity hash function. To compare the accuracy of the current version of PC2LE with our early work, we also consider the PC2LE-FM-1 (Flajolet-Martin with the identity hash function) approach.

MIN-ID: we consider the minimum-id leader election algorithm in Section 4.5 of [Raynal, 2013], extended with our controlled-broadcast optimization (see Section 3.5.1.2). As module identifiers are randomly attributed in the network, this corresponds to the election of a random node.

BARYCENTER: We consider the exhaustive BARYCENTER algorithm presented in [Mamei et al., 2005]. It computes all-pair shortest paths using n simultaneous asynchronous BFSes without acknowledgment. BARYCENTER was proposed as an application of the TOTA tuple-space-based middleware. We use our own implementation of this approach. In our implementation, modules wait for 500 milliseconds after the reception of the last distance update triggered by a BFS message to check for convergence. Note that BARYCENTER does not have a global termination criterion and some nodes can temporarily recognize themselves as centroid.

k-BFS-RAND: We consider our own distributed implementation of the sequential approach [Eppstein et al., 2001] to approximate the node centrality using k BFSes from random nodes. We refer to it as the k-BFS-RAND approach. To be fair in comparison with the k-BFS SumSweep framework, we also fix $k = 10$. In our implementation, every node generates a random number and the k nodes of minimum generated number perform a BFS (ties are broken arbitrarily). Every node estimates its partial farness/eccentricity values using the distance to the k random nodes. In the k-BFS-RAND-SEQ, the BFSes are performed sequentially. The node of minimum generated number is elected as initiator using a variant of the LE_CHEUNG-BFS-ST-CB algorithm. In k-BFS-RAND-PAR, the k BFSes are performed in parallel. All nodes initiate a BFS using a variant of the CHEUNG-BFS-ST-CB algorithm modified with a mechanism to ensure that only the BFSes initiated by the k nodes of minimum generated number terminate (i.e., 10 simultaneous elections). Node identifiers are used to break the ties. Note that k-BFS-RAND-PAR is prone to network congestion because our current version of the controlled-broadcast optimization does not enable to run multiple parallel elections. Once the k BFSes have terminated, the node of minimum centrality value is elected using an STC followed by an STB on the tree rooted at the k^{th} node.

TBCE: We also consider the Tree-Based Center Election (TBCE) algorithm, our own implementation of the election of the node of maximum tree-based centrality [Kim et al., 2013]. We choose this algorithm as it is both time- and memory-efficient.

PC2LE-MC2: The algorithm proposed in [Garin et al., 2012] to estimate node eccentricity is not directly applicable because it targets synchronous distributed systems, because it requires providing an upper bound of the graph diameter and because it does not elect a node but only estimates every node eccentricity value. Thus, to evaluate the performance of the approach proposed in [Garin et al., 2012] in our target system, we use the PC2LE along with the probabilistic counter [Varagnolo et al., 2010] applied in [Garin et al., 2012]. We call this approach PC2LE-MC2 (Maximum-Consensus Counter).

3.9.3.2/ EFFECTIVENESS EVALUATION

In order to exhibit the accuracy of an algorithm, we use the relative center accuracy and the relative centroid accuracy (see Equations (3.16) and (3.17)). We have computed the exact center/centroid and node eccentricity/farness using our tool[5] for external graph analysis.

$$\text{relative centroid accuracy} = 1 - \left| \frac{far(centroid) - far(elected\ node)}{far(centroid)} \right| \tag{3.16}$$

$$\text{relative center accuracy} = 1 - \left| \frac{ecc(center) - ecc(elected\ node)}{ecc(center)} \right| \tag{3.17}$$

Figure 3.15 shows the relative center and centroid accuracy of the different algorithms considered.

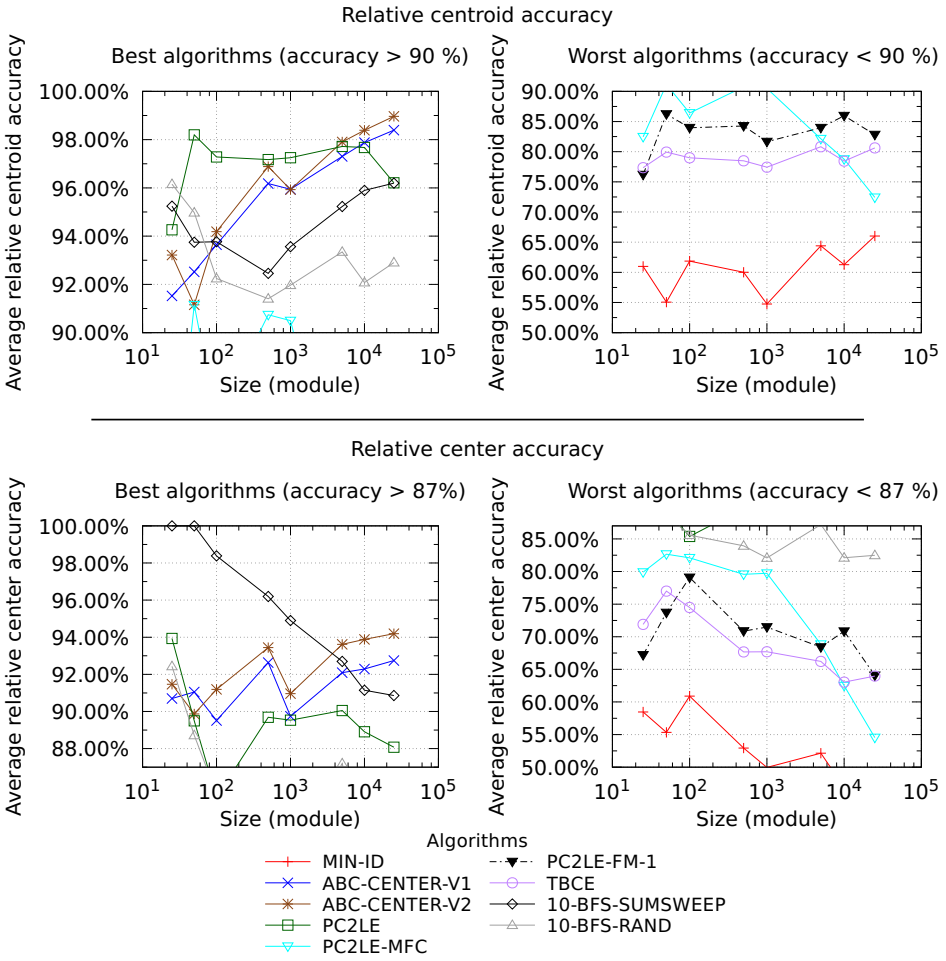


Figure 3.15: Effectiveness of centrality-based leader election algorithms: relative eccentricity and centroid accuracy versus the number of modules in the system. For frameworks, the centroid (resp. center) version is considered for the centroid (resp. center) accuracy.

[5]GraphAnalyzer. Tool available online at: https://github.com/nazandre/GraphAnalyzer

We observe that ABC-Center, PC2LE and k-BFS SumSweep are more accurate than the other algorithms. In systems with 25,000 modules, our algorithms provide a relative centroid accuracy between 96%-99% and a relative center accuracy between 88%-94%. Note that ABC-CenterV2 seems slightly more precise at large scale than the other two.

Furthermore, we observe that performing BFSes from external nodes using the SumSweep heuristic (10-BFS-SUMSWEEP) leads to more accurate results than performing the BFSes from a random sample of nodes (10-BFS-RAND).

Moreover, using the HyperLogLog counter (PC2LE) with the PC2LE framework leads to more accurate results than using the maximum consensus-based probabilistic counter [Varagnolo et al., 2010] used in [Garin et al., 2012] (PC2LE-MC2) and than using the Flajolet-Martin algorithm with a single bitstring, as done in our early work [Naz et al., 2016] (PC2LE-FM-1).

3.9.3.3/ Efficiency Evaluation

In this section, we study the time efficiency, the communication efficiency and the memory usage of the different algorithms.

Simulated Execution Time To measure the execution time, we consider that an algorithm terminates when the node to be elected considers itself elected.

Figure 3.16 shows that the simulated average execution time of all the considered algorithms except BARYCENTER seems to increase linearly with the diameter of the system. The average execution time of BARYCENTER explodes in systems with more than 1,000 modules. We believe that this is due to network congestion.

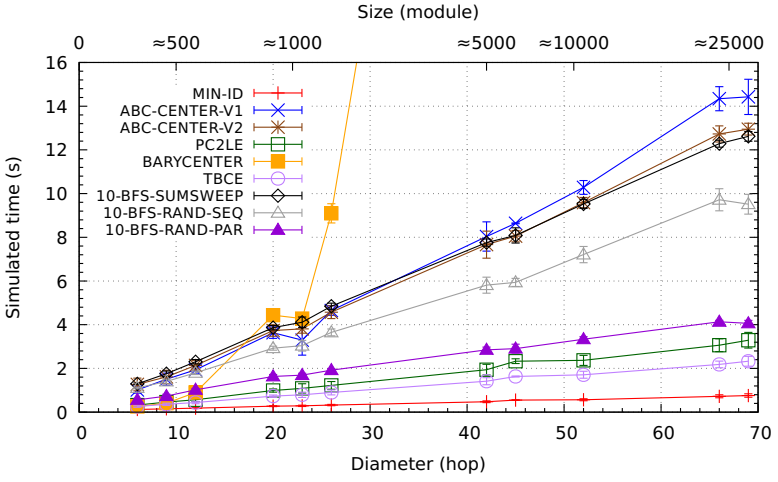


Figure 3.16: Simulated average execution duration (± standard deviation) of centrality-based leader election algorithms versus the system diameter. For each point, at least 5 executions were performed.

ABC-CenterV1, ABC-CenterV2 and k-BFS SumSweep are longer to converge than the other algorithms considered, except for BARYCENTER. Nevertheless, as previously

shown, these algorithms tend to have better center accuracy results than all the others. To give an idea of the convergence time, ABC-Center requires on average 3-4 steps to converge in our systems. Also note that ABC-CenterV2 is slightly faster than ABC-CenterV1.

MIN-ID, TBCE, PC2LE and k-BFS-RAND-PAR scale well in terms of execution time. For Blinky Blocks systems with a diameter of more than 65 hops and a size of approximately 25,000 modules, MIN-ID, TBCE and PC2LE respectively elect a central module in less than 1, 2 and 4 seconds. PC2LE is slightly slower than TBCE and MIN-ID, but is definitely more precise, as shown in the previous section.

Number of Messages Figure 3.17 shows the total number of messages exchanged during the execution of the centrality algorithms considered according to the size of the system. Figure 3.18 shows the average number of messages sent per module. The number of messages used by an algorithm includes all the messages that it generates, even those sent after the final node has been elected. The number of messages sent also reflects the energy consumption of the modules.

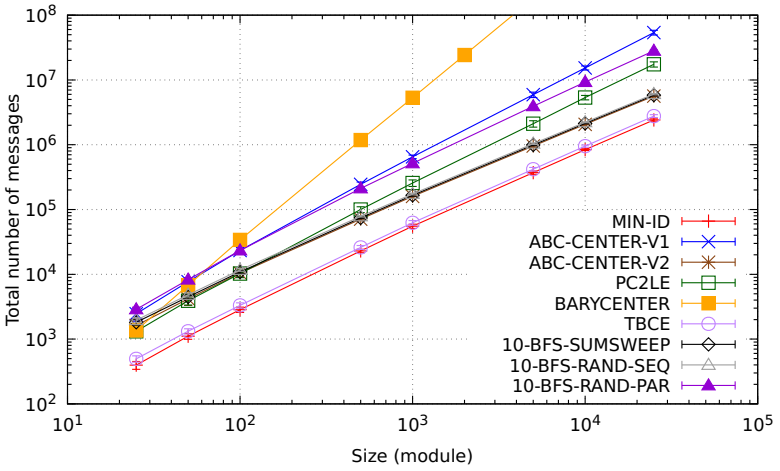


Figure 3.17: Average total number of messages (± standard deviation) of centrality-based leader election algorithms according to the size of the system.

We observe that BARYCENTER uses a lot more messages than the other algorithms. PC2LE tends to use more messages at large scale than ABC-CenterV2 and the sequential k-BFS approaches. Moreover, the latter approaches use more messages than TBCE and MIN-ID. For large-scale systems with 25,000 Blinky Blocks, PC2LE uses about 20×10^6 messages while ABC-CenterV2, 10-BFS-SumSweep, 10-BFS-RAND-SEQ use 6×10^6 messages and TBCE uses only about 3×10^6 messages.

ABC-CenterV1 uses fewer messages than ABC-CenterV2. 10-BFS-SumSweep and 10-BFS-RAND-SEQ approximately use the same number of messages. Notice that 10-BFS-RAND-PAR generates more messages than 10-BFS-RAND-SEQ.

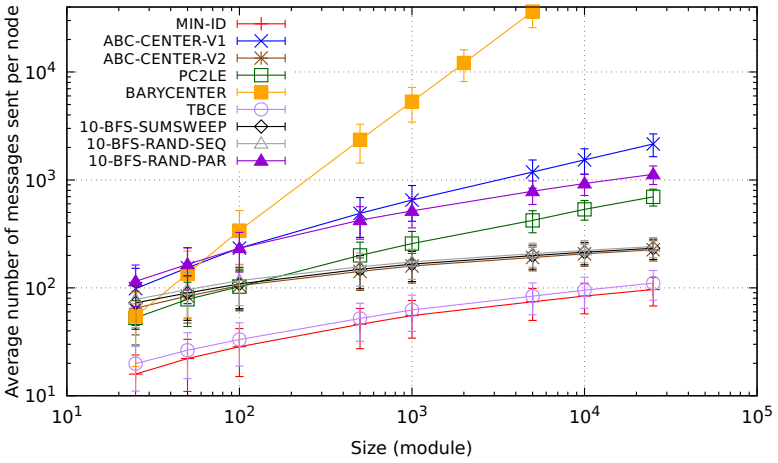


Figure 3.18: Average number of messages sent per node (± standard deviation) of centrality-based leader election algorithms according to the size of the system.

Memory Usage Figure 3.19 shows the maximum memory usage of the different algorithms. The memory usage of an algorithm is composed of its memory footprint, both at the application level and in the different message queues. Note that in the Blinky Blocks firmware, whenever a module broadcasts a message to all its neighbors, a copy of the message is inserted in all its outgoing-message queues. Moreover, the Blinky Blocks store a message using 19 bytes of memory (17 bytes of data and 2 bytes for data related to message handling).

We recall that, in BARYCENTER, every node locally stores $O(n)$ information at the application level and PC2LE stores $O(c + \Delta)$, where c is the cost of the probabilistic counter used. The other algorithms store $O(\Delta)$ information.

ABC-CenterV2, MIN-ID, PC2LE, k-BFS-SumSweep, k-BFS-RAND-SEQ and TBCE scale well in terms of memory usage. In systems with 25,000 nodes, they use less than 500 bytes of memory, among which 380 bytes[6] are due to message queue occupancy. ABC-CenterV1 and 10-BFS-RAND-PAR use up to 10 kbytes in systems with 25,000 modules because of the memory overhead due to message pileups. 10-BFS-RAND-PAR perform BFSes in parallel, thus being faster but requiring much more memory. BARYCENTER uses 600 kbytes in systems with 5,000 modules.

3.10/ DISCUSSION

Electing a central node involves a trade-off between the cost that can be afforded in terms of resources (time, memory, computation, energy) and the desired level of accuracy. Thus the algorithm to be used in order to elect a central node depends on the application, i.e., the role that this central node will play, the stability of the network, the scarcest resource, etc.

[6] $20 \times 19 = 380$ bytes

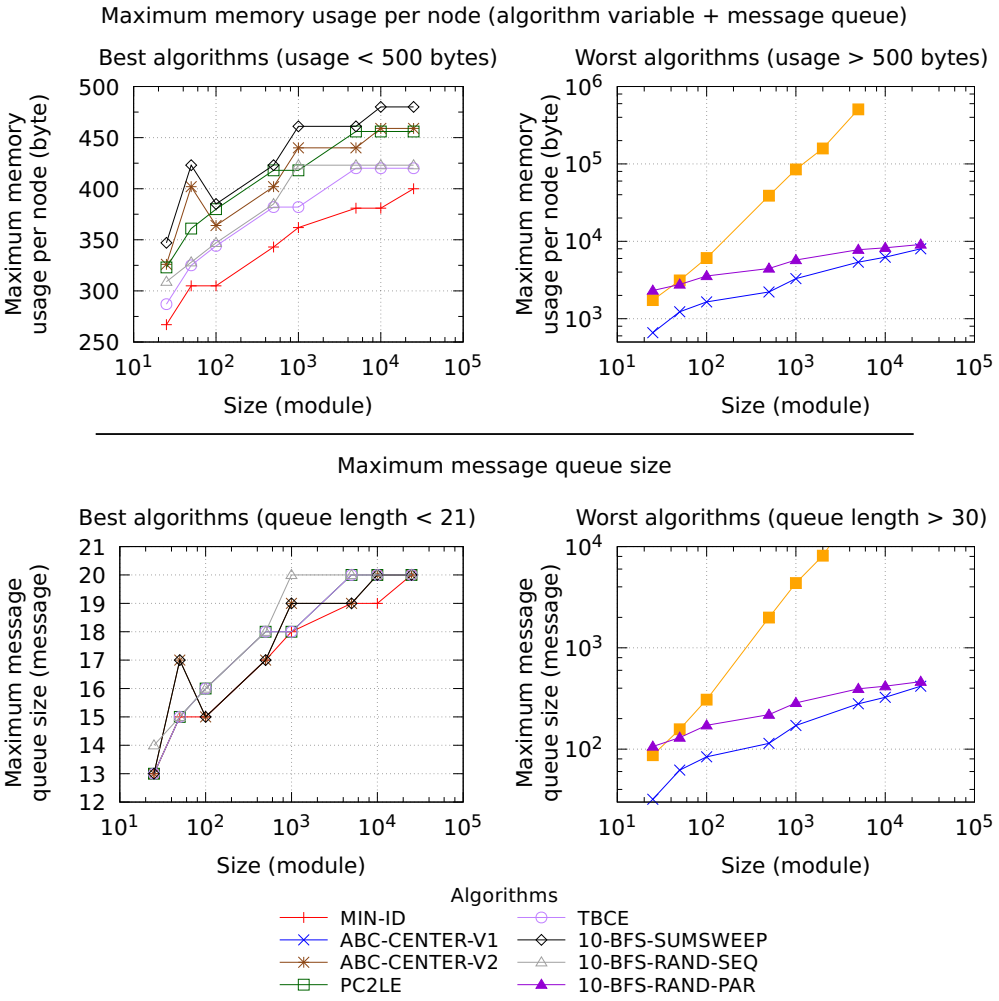


Figure 3.19: Above, the maximum memory usage (considering both the local algorithm variables and the message queue usage) according to the size of the system. Below, the maximum message queue per module (considering both the incoming and outgoing queues).

Exact approaches (e.g., BARYCENTER) are exhaustive and tend to overwhelm the network. They are definitely not suitable for large-scale systems since they are slow to converge, they generate a significant number of messages and may have a large memory footprint. It is paradoxical, since the importance of central nodes increases with the system size. In 5,000 node systems, BARYCENTER requires nearly 45 seconds to converge and uses more than 500 kbytes per node.

Electing a random node using MIN-ID leads to poor accuracy but scales well in terms of efficiency. TBCE provides a better accuracy while being only slightly slower and using a similar number of messages. In systems of 25,000 modules, TBCE runs on average in 2.3 seconds and has a relative centroid (resp. center) accuracy of 81% (resp. 64%).

We proposed PC2LE which is slightly slower than TBCE but is definitely more accurate. In systems of 25,000 modules, PC2LE runs in 3.3 seconds and provides a relative centroid accuracy of 96% and a relative center accuracy of 88%. However, this better accuracy

comes at the price of a higher message cost.

We proposed ABC-CenterV2 and k-BFS-SumSweep which are the most accurate center approximation algorithms. They perform BFSes from specific nodes, which leads to more accurate results than computing BFSes from random nodes as in k-BFS-RAND. In 25,000 Blinky Blocks systems, ABC-CenterV2 elects, on average, a 99% accurate centroid and a 94% accurate center. ABC-CenterV2 and the k-BFS-SumSweep are, however, slow to converge as the BFSes are performed consecutively. In 25,000 module systems, they run in almost 13 seconds. These two algorithms use more messages than PC2LE and MIN-ID but less messages than PC2LE.

BFSes cannot be parallelized in ABC-CenterV2 and k-BFS-SumSweep, but if it was possible, naively performing BFSes in parallel would overwhelm the network and incur a large memory overhead. Indeed, k-BFS-RAND-PAR, which performs k BFSes in parallel, uses at most 10 kbytes per node, while k-BFS-RAND-SEQ, in which the k BFSes are computed consecutively, only uses 423 bytes.

ABC-CenterV2, k-BFS SumSweep, MIN-ID, TBCE and PC2LE all have a limited memory cost. They use between 400 and 480 bytes per node max.

3.11/ CONCLUSION

In this chapter, we proposed a collection of efficient and effective distributed algorithms to elect approximate-centroid and approximate-center nodes in asynchronous distributed systems. We evaluated our algorithm on the Blinky Blocks modular robotic system, using both hardware experiments and simulations. Results show that our algorithm scales well in terms of accuracy, execution time, number of messages and memory usage. To the best of our knowledge, our algorithms are the most precise existing distributed algorithms dedicated to the election of an approximate centroid or an approximate center in our target systems, with both a reasonable convergence time and a limited storage cost.

In the next chapter, we study time synchronization in LMRs. We use the algorithms proposed in this chapter to elect a central node that synchronizes all the others. As shown in the Introduction section of this chapter, using a central module rather than a random one leads to more precision.

4

TIME SYNCHRONIZATION

Contents

4.1/ INTRODUCTION

In modular robotic systems, coordination among a group of modules often relies on the existence of a common notion of time. For instance, in the conveyance surface presented in Section 2.2.3, modules cooperate to convey the object using distributed real-time control. They have to remain synchronized in order to satisfy timing constraints, otherwise the object may get out of the trajectory, hit obstacles or fall off the surface. The next section presents another interesting application, the distributed bitmap scroller, in which every module is a pixel and the modules collaboratively scroll a bitmap in a synchronous way. Coordination of the modules requires synchronized clocks. More generally, many applications that involve distributed control and actuators need a common notion of time.

Modules can share a common timing signal through dedicated pins, but this requires a specific hardware design. In this chapter, we consider a system without a global clock signal. Every module has its own notion of time provided by its own hardware clock. Since common hardware clocks are imperfect, local clocks tend to run at slightly different and variable frequencies, drifting apart from each other over time. Consequently, a distributed time synchronization is necessary to keep the local clock of each module synchronized to a global timescale. The offset of two clocks denotes the time difference between them, whereas the skew between two clocks denotes their frequency difference.

Network-wide synchronization protocols aim to keep a small offset between local clocks and a global reference time. In most of the existing protocols, devices exchange timestamped messages in order to estimate the current global time. Since time keeps going during communications, modules have to correctly compensate for network delays in order to evaluate the current global time upon reception of synchronization messages. Although it is non-trivial to accurately estimate communication delays, especially in the presence of unpredictable delays (due, for example, to queueing or retransmissions), it is crucial in order to achieve high-precision performance.

The contribution of this chapter is to propose the Modular Robot Time Protocol (MRTP), a network-wide time synchronization protocol for modular robots with neighbor-to-neighbor communications. MRTP is intended to synchronize fairly stable systems where changes in the network topology, due for instance to module mobility, or potential module or link failures, are infrequent. We assume that every module has a local clock, which can be low-precision and low-resolution, typically in the order of the millisecond. Furthermore, modules can use low communication bitrates (e.g, 38.4 kbit/s). In addition, we assume that modules can timestamp messages at the data-link layer. Such a low resolution, low precision and high communication latency make accurate synchronization challenging. First, the local time cannot be accurately read. Second, it is hard to accurately compensate for network delays if they are not negligible and, at the same time, only roughly measurable. Third, clock skew and clock instability may not be negligible during high-latency (multi-hop) communications.

To the best of our knowledge, MRTP is the first protocol for modular robots that provides an accurate low-skew global timescale without dedicated hardware. Our protocol combines new ideas with existing methods proposed in the domains of computer networks

and wireless sensor networks. In our protocol, a dynamically elected central module periodically broadcasts the current global time along the edges of a spanning tree. Placing the time master close to the center of the system reduces the time of the synchronization phases and increases the overall precision as cumulative estimations are made every hop. The method to compensate for communication delays is carefully chosen, depending on the target systems. In Blinky Blocks systems, we use data-link layer timestamping and predictions of the transfer time (as defined in Section 4.4.2) to correctly compensate for network delays. A module gets synchronized by a single timestamped message from its parent one level higher in the tree, incurring little message overhead. Furthermore, modules use linear regression to compensate for clock skew.

We implemented our protocol and evaluated it on the Blinky Blocks system, both on hardware[1,2] and in the VisibleSim simulator[3] (see Section 2.3). We show that MRTP is able to manage systems composed of up to 27,775 Blinky Blocks. Furthermore, experimental results show that MRTP is capable of successfully maintaining a Blinky Blocks system synchronized to a few milliseconds, using few network resources at runtime, although the Blinky Blocks use 38.4 kbit/s communications and are equipped with very low accuracy (10,000 parts per million (ppm)) and poor resolution (1 millisecond) clocks.

The rest of this chapter is organized as follows. Section 4.2 presents a practical application of MRTP in order to motivate our work and to show its necessity. Section 4.3 offers an overview of the existing time synchronization protocols. Section 4.4 details the system model and assumptions. Section 4.5 describes MRTP. Section 4.6 describes the technical characteristics of the Blinky Blocks, i.e., the target platform. Section 4.7 presents experimental results. Section 4.9 concludes our work.

4.2/ EXAMPLE OF APPLICATION: THE DISTRIBUTED BITMAP SCROLLER

This section presents the distributed bitmap scroller application[4]. In this application originally imagined by Benoît Piranda, every module represents a pixel and the modules cooperatively scroll a text (here “Femto-st”) using color changes. The scroller is extensible and robust to system split and merge. Figure 4.1 shows a distributed bitmap scroller made from 72 Blinky Blocks. We first present our implementation and then discuss the need for global time synchronization.

[1] The source code of MRTP is included in the Blinky Blocks firmware, available online at https://github.com/claytronics/oldbb

[2] Some examples of MRTP running on the Blinky Blocks platform are available online in video at https://youtu.be/66D12ESGc98 and https://youtu.be/X6QzivsmJBo

[3] The source code of VisibleSim and the applications written for the evaluation of our protocol are available online at: https://github.com/nazandre/thesis

[4] A video of a distributed bitmap scroller made from 72 Blinky Blocks synchronized using MRTP is available online at https://youtu.be/66D12ESGc98

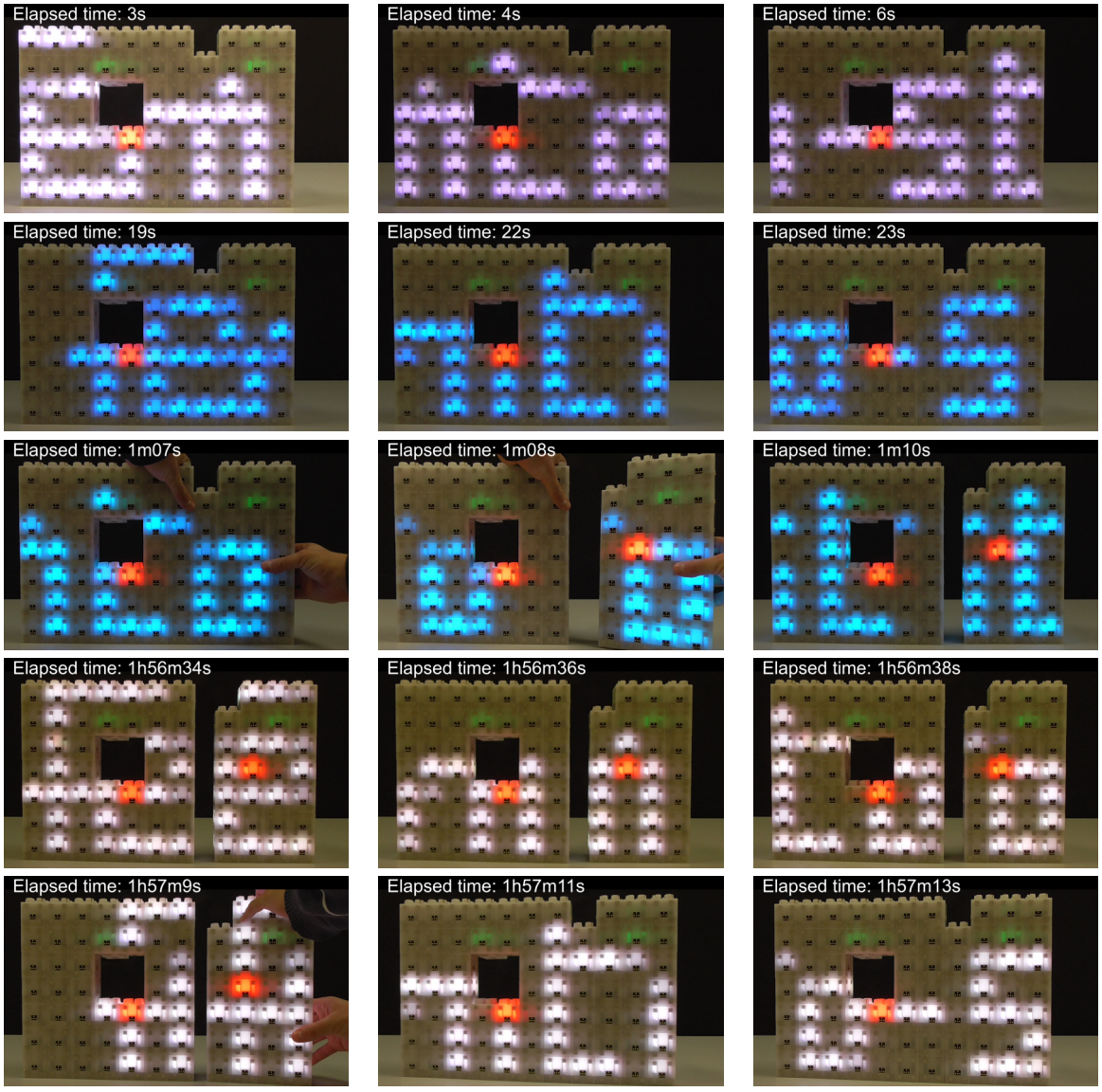


Figure 4.1: A distributed bitmap scroller made from 72 Blinky Blocks. The system scrolls “Femto-st” in different colors. The blocks are synchronized using MRTP. The time master stays in red.

4.2.1/ Our Implementation

In our implementation, modules first distributively build a coordinate system. Then, they start to display the text that is shifted one column to the left every 250 milliseconds. From a local point of view, every module stores the global bitmap to display and locally updates its color on a regular basis, based on the module position and on its current clock time, only. The vision persistence is around 40 milliseconds. Hence, when the text is shifted one column to the left, all modules should change their color within 40 milliseconds in order for the color changes to appear synchronized. In our implementation, the system is globally synchronized using MRTP. The time master stays in red.

As shown in Figure 4.1, the bitmap scroller and thus MRTP are robust to system merge

and split.

4.2.2/ Need for Global Time Synchronization

In order to show that the distributed bitmap scroller requires a global timescale, we subsequentially discuss the issues risen by a non-exhaustive list of alternative approaches.

Unsynchronized scroller Figure 4.2 shows our implementation of the bitmap scroller running without time synchronization. Because of clock skew, module clocks progressively drift apart from each others causing the modules to light asynchronously and the text being scrolled to become unreadable. Hence, individual color changes need to be synchronized in order to ensure a synchronous scrolling at the global scale.

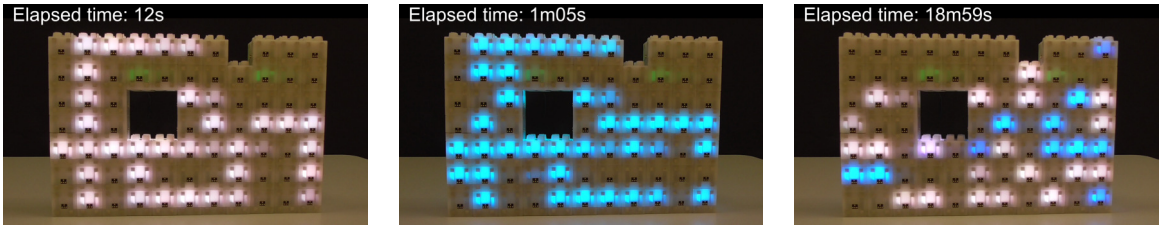


Figure 4.2: Unsynchronized bitmap scroller of 72 Blinky Blocks.

Centralized or Distributed Control based on Order Propagation A different approach than clock synchronization is to use color change orders to control the system and to dictate the pace of the text shifting.

After having locally observed a delay of 250 ms, a single elected module, or any module, can flood a message to request all the modules to update their color upon reception. However, immediate-term order propagation relies on fast propagation. In our example, a Blinky Block sends a message to a neighboring module on average in 6 milliseconds. If we do not consider message time of residence at nodes, a message needs at least 42 milliseconds to travel over seven hops. Hence, after seven hops, a delay in color changes will be observed and the one-column text shifting will appear unsynchronized.

Color updates can also be scheduled to a future date at which all modules will have received the information. However, it is difficult to predict that time. Indeed, order propagation may be delayed due to the network load for instance. Moreover, it is not possible to precisely schedule a global event too far in the future because of hardware-clock imprecision (skew, noise, etc.).

Moreover, this approach is less robust to message loss than the clock synchronization approach. If an order message is lost, then some modules will not update their color for a step. On the other hand, in our implementation, if an MRTP synchronization message gets lost, all modules will still update their color, but with a slight delay.

Right-to-Left Pixel Propagation In this approach, a module holds for 250 milliseconds the current pixel that it has to display. Pixels are propagated from the left to the right using messages to produce column shifts. The right-most module of every row is responsible to start displaying a given pixel. We name these modules the pixel initiators.

An immediate limitation of this approach is that it requires some routing procedure in the presence of holes. Indeed, the left-next module may not be an immediate neighbors. Moreover, this approach less robust to failures than the previous one. Indeed, a pixel gets lost if the message that carries it is lost.

Regarding synchronization, pixel initiators have to be synchronized in order to synchronously start the propagation of the pixels. However, even-though pixel initiators are synchronized, delays in color updates may still be observable. As every module has its own notion of time, pixels will reside at modules for slightly different durations. Hence, pixels will not propagate at the exact same speed in all rows, causing color changes to become more and more unsynchronized with the hop distance.

Limited-scope Time Synchronization One may envision to synchronize only neighboring modules together. As shown in the Section 4.7.2, with this approach, modules are not well synchronized to a global time in large-scale systems. Hence, delays will be observed in the color changes.

Alternatively, one can also envision to synchronize all modules of a same column together. However, because of clock skew, columns will progressively drift apart from each others and color changes will not appear to be all synchronous. Moreover, this approach may be tricky to implement in the presence of holes.

Hence, limited-scope synchronization is not sufficient. In the distributed bitmap scroller, all modules have to synchronously perform an action (i.e., update their color). Hence, all clocks should be synchronized to a global timescale.

4.3/ State of the Art

Time synchronization has been extensively studied in various domains. Many algorithms and protocols have been proposed for computer networks such as Cristian's algorithm [Cristian, 1989], the Berkeley algorithm [Gusella et al., 1989], the Network Time Protocol (NTP) [Mills, 1991] and the IEEE 1588 Precise Time Protocol (PTP) [IEEE, 2008]. Time synchronization is also an important topic of interest in Wireless Sensor Networks (WSNs) where many protocols have been proposed, e.g., Reference Broadcast Synchronization (RBS) [Elson et al., 2002], the Timing-sync Protocol for Sensor Networks (TPSN) [Ganeriwal et al., 2003], the Flooding Time Synchronization Protocol (FTSP) [Maróti et al., 2004], the Time-Diffusion Synchronization Protocol (TDP) [Su et al., 2005], the Rapid Time Synchronization (RATS) [Kusy, 2007], the PulseSync [Lenzen et al., 2009, Lenzen et al., 2015], the Asynchronous Diffusion algorithm (AD) [Li et al., 2006], the Gradient Time Synchronization Protocol (GTSP) [Sommer et al., 2009], the

Average TimeSynch (ATS) protocol [Schenato et al., 2011] and the Maximum Time Synchronization (MTS) [He et al., 2014a]. Like modular robots, WSNs generally form spontaneous peer-to-peer networks of resource-constrained devices. To the best of our knowledge, time synchronization has not attracted any attention in the modular robotic community. Methods to provide a global metronome-like signal in modular robots have been proposed in [Kokaji et al., 1996, Baca et al., 2010]. However, these mechanisms synchronize clock phase or/and frequency, but not actual clock time. Moreover, [Kokaji et al., 1996] is purely theoretical, the authors consider ideal clocks running at the same exact frequency and do not provide any performance evaluation. In [Stoy, 2003, Stoy et al., 2002b, Stoy et al., 2002a], the authors propose the role-based distributed control algorithm for modular robotic systems. It enables to coordinate module actions in order to produce a global behavior. In this method, a periodic logical signal is established in the system using message passing (e.g., a sine wave signal to produce a caterpillar-like locomotion in a chain of modules). However, this control method does not establish a global timescale and ignores communication delays.

4.3.1/ Architecture : from Master/Slave to fully Distributed Protocols

Existing time synchronization protocols differ by the network architecture they adopt. NTP, PTP, TPSN, FTSP, PuleSync, RBS and TDP adopt a master/slave approach. In a master/slave approach, one or more masters are in charge of synchronizing slave nodes. In NTP, PTP, TPSN, FTSP, PuleSync and TDP, slave node clocks are adjusted to a reference time held by the time master(s). The reference time can be the Coordinated Universal Time or the master local clock. In the Berkeley algorithm, slave node clocks are adjusted to an aggregated value of some or all the system clock values. These approaches aim at performing global synchronization, i.e., keeping all nodes synchronized together. These protocols provide a satisfactory synchronization precision between arbitrary nodes but may poorly synchronize neighboring nodes. This is due to the fact that two neighboring nodes can be synchronized by messages that have traveled on long and almost independent paths, causing the error accumulated at every hop to be propagated differently.

In contrast, AD, ATS, GTSP and MTS are fully distributed. In these protocols, nodes exchange timing information with all their one-hop neighbors on a regular basis. In AD, every node frequently adjusts its clock to the average value of its neighbors' clock. ATS and GTSP use a similar consensus-based averaging technique. These average-based approaches primarily aim at achieving local synchronization, i.e., keeping neighboring nodes synchronized together, allowing nodes to have a larger pairwise synchronization error with nodes that are faraway. MTS and its variants proposed in [He et al., 2014a, He et al., 2014b] use extremum-value-based consensus to achieve faster convergence. In general, fully distributed methods are naturally fault-tolerant and robust to node mobility. However, they can lead to a long convergence time and to a high message complexity, especially in point-to-point networks without broadcast support. Indeed, in systems without local or global shared broadcast medium, a node has to send individual messages to

all neighbors in order to broadcast messages.

4.3.2/ INFRASTRUCTURE OF MASTER/SLAVE PROTOCOLS

Master/slave time synchronization protocols differ by the infrastructure they use. Protocols can use tree-like structures, cluster-based structures or be infrastructure-less.

Tree-like Structures NTP, PTP and TPSN use tree-like hierarchical structures rooted at the time master(s) to spread timing information. Logical neighbors in the tree(s) can be neighbors in the physical network as in TPSN, or potentially distant as in NTP. The latter case may require multi-hop communications that rely on the existence of an underlying routing service. In our case, we assume no routing service. In TPSN, nodes are recursively synchronized hop-by-hop along the edges of the synchronization tree starting from the time master. Hence, during each synchronization phase, the current global time gets quickly disseminated through the entire network. In addition to providing a relatively quick synchronization convergence, this reduces the impact of clock inaccuracies (due to noise, skew variations, time-increasing errors in the local estimation of the global time) on the synchronization process.

Clustering based on Broadcast Domains In RBS, nodes maintain relative timescales of their neighborhood using reference pulses broadcast by some master nodes. In multi-hop networks, nodes can be grouped into overlapping clusters based on broadcast domains and border nodes act as gateways to translate clock values.

Infrastructure-less Approaches In contrast, FTSP, RATS and PulseSync are infrastructure-less. They provide robustness to network topology changes and to link failures using either periodic local broadcasts or periodic network-wide floodings. In FTSP, the time master and the synchronized nodes periodically broadcast their estimation of the current global time to all their neighbors, in an asynchronous way. Synchronization waves propagate with a limited speed through the network. Indeed, after having received a new synchronization message, a node has to wait until the expiration of its broadcast period to transmit the information to its neighbors. As a consequence, the time-increasing estimation error of the global time is amplified at every hop and FTSP exhibits a synchronization error that grows exponentially with the size of the network [Lenzen et al., 2009]. Hence, optimal synchronization requires fast network flooding [Lenzen et al., 2009]. RATS and PulseSync employ rapid network-wide floodings using recursive broadcasts to quickly disseminate the global time through the network. The time master periodically launches synchronization waves using broadcasts. Slave modules re-broadcast new synchronization messages shortly after reception. In [Ferrari et al., 2011], the authors propose a sophisticated mechanism to provide fast network flooding in IEEE 802.15.4 WSNs and thus accurate time synchronization. In reliable and fairly static point-to-point networks without broadcast support, recursive synchronizations using a tree-like structure are more

communication-efficient than network-wide flooding.

4.3.3/ Communication Delay Compensation Methods

Time synchronization protocols also differ by the methods they use to compensate for communication delays. The method to be applied depends on the target platform and more precisely on the communication mechanism and the precision with which time can be measured. This choice directly impacts the precision of the synchronization protocol. Existing methods can be divided into three categories: approaches based on the round-trip time, methods based on byte-level timestamping and approaches based on reference broadcasts.

Round-Trip Time based Methods Cristian's algorithm, the Berkeley algorithm, NTP, PTP, TPSN and TDP measure half the round-trip time to estimate one-way communication delays. Cristian's algorithm and NTP perform end-to-end synchronization on possibly multi-hop paths. They use statistical analysis to mitigate variations in delays due to retransmission(s), queueing, route selection, etc. These methods are expensive in communications and in computations. PTP and TPSN propose to perform per-hop synchronization with low-level timestamping to prevent unpredictable delays induced at the different layers of the network stack from affecting delay measurements. PTP can use timestamps recorded at the physical layer to achieve high accuracy if dedicated hardware is available. In TPSN, a node exchanges a single bidirectional message timestamped at the boundary of the data-link layer to synchronize itself to another node and compensates for communication delays using half the round-trip time. We call this method RTT (for Round-Trip Time). Round-trip time methods assume symmetrical nominal delays and usually neglect the effect of clock skew during the round trip. In [Syed et al., 2006], Syed et al. study time synchronization in underwater acoustic sensor networks where propagation times of several hundred milliseconds are observed. They propose to use skew-compensated two-way message exchanges in these systems.

Byte-level Timestamp based Methods FTSP, PulseSync and the practical implementations of both ATS [Schenato et al., 2011] and MTS [He et al., 2014b] use byte-level time-stamping, which requires an intimate access to the data-link layer.

In the last two methods, nodes exchange a single unidirectional message timestamped just before the transmission of the first byte (i.e., the Frame Delimiter byte) and upon reception of this byte. The time elapsed between the transmission and the reception of the frame delimiter byte is neglected and ATS / MTS consider that the two timestamps refer to the same real time. We call this method FD (for Frame Delimiter). This method neglects the interrupt handling time, the frame delimiter byte transmission / reception, the propagation time and the time required for the detection of the frame delimiter byte. Although FD works well in low-latency networks, the neglected time can be important in higher-latency systems. For instance, our target system uses 38.4 kbit/s connections while WSNs that use IEEE 802.11b communications have a maximal bitrate of 11 Mbit/s.

At 38.4 kbit/s, a byte is transmitted in roughly 208 μs, while at 11 Mbit/s a byte is sent in less than 1 μs.

FTSP goes one step further in order to eliminate most of the sources of delays in message transmission (except for the propagation time). FTSP synchronizes neighbors using a single message broadcast with statistical operations on timestamps captured at the byte boundary during interrupts at the data-link layer. The latest version of PulseSync [Lenzen et al., 2015] is based on an enhanced version of the FD method. The authors use the slotted programming approach [Flury et al., 2010] to minimize the interrupt latency and use a static value measured experimentally during a calibration phase to compensate for the time between the insertion of the timestamp, just before transmitting the frame delimiter byte, and its detection upon reception. However, the method proposed in FTSP and PulseSync cannot be applied directly to our target system. Indeed, we assume low-resolution clocks, typically in the order of the millisecond, that cannot efficiently capture phenomena at the byte transmission level which occurs on the microsecond scale.

Reference Broadcast In RBS, some reference nodes periodically broadcast reference messages. Neglecting propagation delays, receiving nodes use the data-link reception times as reference points to compare their clock values all together. This requires a shared broadcast medium and it is not usable in point-to-point networks.

Discussion We argue that the method of compensating for communication delays has to be selected as a function of the target system. If we assume a predictable transfer time between neighbor modules, we propose to perform per-hop synchronization using a single unidirectional message timestamped at the data-link layer and predictive communication delay compensation (see Section 4.4.3). We call this method PRED (for Predictive). We show in the evaluation section 4.7.1.3 that, in our target system, PRED is on average more precise than the other two methods that can be applied to our target system, namely FD and RTT. Note that this is mainly due to the fact that the average transfer delay of a frame is almost a round number (on the millisecond scale) in this system.

4.3.4/ Clock Model: from Clock Offset Adjustment only to Clock Skew Compensation

Furthermore, time synchronization protocols differ in the clock model they use. In some protocols, e.g., AD and TPSN, nodes perform clock offset adjustment only and do not take into account clock skew. Compensation for clock skew enables modules to be synchronized less frequently without degrading the synchronization precision.

NTP uses phase-locked loops and/or frequency-locked loops. In [Kim et al., 2012], the authors use a Kalman filter to track clock offset and skew with low-precision oscillators and time-varying skew. Indeed, in the presence of ambient environment variations (e.g., temperature variations), the clock skew may vary over time.

ATS, Belief Propagation (BP) [Etzlinger et al., 2014], GTSP, Mean Field (MF) [Etzlinger

et al., 2014], MTS, FTSP, RBS, PulseSync, RATS and [Noh et al., 2007, Leng et al., 2010] propose to model clock using a linear model computed from recent observations, assuming that oscillators have high short-term stability. Indeed, if we assume that environment changes do not happen or happen gradually, the clock skew will change smoothly. RBS, FTSP, PulseSync and RATS use least-square linear regression on a recent window of observations. ATS and GTSP use an averaging technique to estimate the clock skew based on the previous synchronization point. In [Noh et al., 2007, Leng et al., 2010], the authors propose to enhance TPSN by using a linear model and maximum likelihood estimators. BP and MF derive maximum a posteriori estimators of the clock parameters using belief propagation and mean field on factor graphs, respectively. Different methods for clock skew compensation including linear regression, exponential averaging and phase-locked loops have been evaluated in [Amundson et al., 2008]. Although results are nearly identical, experiments suggest that linear regression leads to slightly more precision.

Note that, in addition to compensating for clock skew, these aggregating techniques also tend to reduce the impact of the measurement errors due to the resolution of the timestamps.

4.3.5/ Time Master Election

Master/slave time synchronization protocols also differ by the mechanisms they employ to select the time master. In NTP and RATS, time masters are pre-configured. In our case, it is more flexible if the system itself elects its time master. In PTP and TDP, elections are based on the quality of the clocks. In addition, TDP periodically re-elects time masters to balance the load. FTSP and PulseSync implicitly elect the minimum-identifier node as the time master during the synchronization phases.

In our case, we consider systems where all modules are identical and equipped with the same hardware clocks. Although these clocks differ slightly in their accuracy and stability, we consider that with a careful selection of the hardware, the impact of cumulative errors in network delay estimations will be predominant in large-diameter systems. A random error is experienced at each hop. Let us assume that these per-hop errors are independent and identically distributed with a mean of λ and a standard deviation of σ. The Central Limit theorem states that the error accumulated over k hops follows a normal distribution with a mean of λk and a standard deviation of $\delta\sqrt{k}$. Experimental results presented in Section 4.7.1.3 confirm this trend. Hence, we propose to elect a central module as the time master.

4.3.6/ Summary

Table 4.1 summarizes the related work. Existing protocols contain interesting ideas but fail to efficiently adapt to homogeneous modular robot systems where modules use low-bitrate neighbor-to-neighbor communications, hardware clocks have low precision and the network diameter can be large. In the absence of a (locally) shared communication medium, infrastructure-less approaches are too expensive in terms of communication in

Name	Domain	Architecture	Infrastructure	Synchronization Technique	Clock Skew Compensation
NTP [Mills, 1991]	Computer Networks	Master/Slave Master(s): pre-configured	Tree	(Multi-hop) round-trip messages with frame-level timestamps and statistics	Phase-locked and/or frequency-locked loops
PTP [IEEE, 2008]	Computer Networks	Master/Slave Master: clock quality based election	Tree	Round-trip messages with low-level (data-link to physical layer) timestamps and per-hop delay compensation	
TPSN [Ganeriwal et al., 2003]	Sensor Networks	Master/Slave	Tree	Recursive per-hop synchronization. Round-trip messages with frame-level timestamps	/
TPSN + MLE [Leng et al., 2010]	Sensor Networks	Master/Slave	Tree	Recursive per-hop synchronization. Round-trip messages with frame-level timestamps and statistics	Linear model with maximum likelihood estimators
TDP [Su et al., 2005]	Sensor Networks	Masters/Slave multiple changing masters: clock quality based election	/	Recursive per-hop synchronization. Bidirectional round-trip messages with statistics	/
RBS [Elson et al., 2002]	Sensor Networks	Master/Slave	Broadcast-domain based clustering	Reference broadcast	Linear model with least-square linear regression
FTSP [Maróti et al., 2004]	Sensor Networks	Master/Slave Master: id-based implicit election	/	Periodic asynchronous broadcasts. Unidirectional broadcast with byte-level timestamps and statistics	Linear model with least-square linear regression
RATS [Kusy, 2007]	Sensor Networks	Master/Slave Master: pre-configured	/	Recursive per-hop synchronization. Unidirectional broadcast with byte-level timestamps and statistics	Linear model with least-square regression
Pulse-Sync [Lenzen et al., 2009, Lenzen et al., 2015]	Sensor Networks	Master/Slave Master: id-based implicit election	/	Recursive per-hop synchronization. Unidirectional broadcast with byte-level timestamps and statistics	Linear model with least-square linear regression
AD [Li et al., 2006]	Sensor Networks	Fully distributed	/	Average-based consensus	/
GTSP [Sommer et al., 2009]	Sensor Networks	Fully distributed	/	Average-based consensus. Unidirectional broadcast with byte-level timestamps and statistics	Linear model with an averaging technique
ATS [Schenato et al., 2011]	Sensor Networks	Fully distributed	/	Average-based consensus. Unidirectional broadcast with byte-level timestamps	Linear model with an averaging technique
MTS and its variants [He et al., 2014a, He et al., 2014b]	Sensor Networks	Fully distributed	/	Extremum-value based consensus. Unidirectional broadcast with byte-level timestamps	Linear model with possibly an averaging technique
BP and MF [Etzlinger et al., 2014]	Sensor Networks	Master/Slave or fully distributed	/	Belief propagation and mean field. Single-hop bidirectional messages with frame-level timestamps	Linear model with maximum a posteriori estimators
Our Contribution: MRTP	Modular Robotic	Master/Slave Master: centrality-based election	Tree	Recursive per-hop synchronization. Selection of the most suited communication delay compensation method for the target system	Linear model with least-square linear regression

Table 4.1: Summary of the state of the art on time synchronization.

compact systems compared to tree-based approaches. The method to compensate for network delays has to be carefully selected in function of the target platform. Furthermore, criteria considered for time master election are not adapted to modular robots running under our assumptions. Node centrality can be considered for the election in order to increase the overall synchronization precision.

4.4/ System Model and Assumptions

In this chapter, we consider modular reconfigurable robots that form asynchronous non-anonymous point-to-point connected networks in which modules use neighbor-to-neighbor communications. We assume that every module has a unique identifier and maintains a consistent list of its neighbors. Furthermore, our protocol is intended to synchronize fairly stable systems where changes in the network topology, due for instance to module mobility, or potential module or link failures, are infrequent. A modular robot can be modeled by an undirected and unweighted graph of interconnected entities $G = (V, E)$, with V the set of vertices representing the modules, E the set of edges representing the connections, $|V| = n$, the number of vertices and $|E| = m$, the number of edges. We use the general graph theory concepts such as the distance between two nodes and the diameter d of the graph.

4.4.1/ Clocks: Notation and Assumptions

Each module M_i is equipped with its own internal clock and has its own local time $L^{M_i}(t)$, an approximation of the real time t. The goal of MRTP is to maintain a global timescale $G(t)$ across the system. We denote $G^{M_i}(t)$, the estimation of $G(t)$ of the module M_i. MRTP preserves time monotonicity and prevents time from running backward, i.e., for any module M_i, $\forall(t, t'), t \geq t', G^{M_i}(t) \geq G^{M_i}(t')$. Moreover, we consider clocks which have high short-term frequency stability but which can be low-precision and can have high skew with respect to one another. Such clocks tend to drift apart from each other in a quasi-linear way over a short period of time.

We consider two synchronization error metrics. We define the module M_i relative synchronization error with respect to the global time at real time t as:

$$\epsilon^{M_i}(t) = G^{M_i}(t) - G(t) \tag{4.1}$$

We define the maximum pairwise synchronization error at real time t, $\epsilon(t)$, as the maximum difference between any two global clocks in the system:

$$\epsilon(t) = \max_{M_i, M_j} \left| G^{M_i}(t) - G^{M_j}(t) \right| \tag{4.2}$$

Since our goal is to achieve global synchronization, we do not consider local synchronization error metrics such as the maximum pairwise synchronization error between neigh-

boring nodes [Lenzen et al., 2009].

4.4.2/ Sources of Network Delays

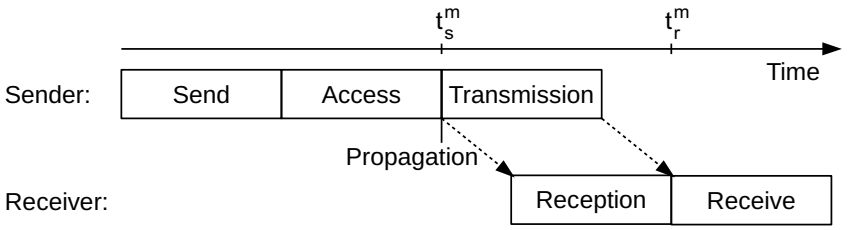


Figure 4.3: Sources of delivery delays in the exchange of a message m between two neighbor modules.

As indicated in [Ganeriwal et al., 2003, Maróti et al., 2004, Amundson et al., 2008], the exchange of a single message m between two neighbor modules can be typically characterized by the steps presented in Figure 4.3. Sending and receiving times represent the times necessary for the message to travel from the application to the data-link layers. These delays are introduced by the operating system and are highly non-deterministic. The access time represents the waiting time at the data-link layer for accessing the communication channel. This time is also highly non-deterministic. The transmission and reception times represent, respectively, the times to transmit and to receive the frame using a bit-by-bit transmission at the physical layer. These delays are mainly deterministic and depend on the length of the frame and the bitrate. The propagation time represents the time necessary for the bits to travel from the sender to the receiver over the physical link. This delay is highly deterministic and depends on the distance between the modules involved in the communication and on the propagation speed over the physical link. We define the transfer time, $T^m_{transfer}$, as the sum of the transmission, propagation and reception times for a message m. These times are highly deterministic.

4.4.3/ Predictive Method to Compensate for Communication Delays

We propose to use the predictive method (PRED) to compensate for communication delays whenever they can be predicted. PRED is a naive method that relies on the assumption that $T^m_{transfer}$ is predictable with a certain accuracy that directly impacts the precision of our protocol. Moreover, it assumes that messages can be timestamped at the data-link layer, shortly before the beginning of the transmission at time t^m_s and upon complete reception at time t^m_r. If we neglect the interrupt handling time, $T^m_{transfer} = t^m_r - t^m_s$.

To compensate for communication delays, the predictive method (PRED) works as follows: Let us assume that a module M_i receives a message m from a module M_j and that m has been timestamped at the data-link layer on both sides (i.e., m contains $L^{M_j}(t^m_s)$ and $L^{M_i}(t^m_r)$). Then, the module M_i can compensate for the communication delays of m and

estimate the local time of M_j at the reception of m by:

$$L^{M_j}(t_r^m) \approx L^{M_j}(t_s^m) + T_{transfer}^m \tag{4.3}$$

4.5/ THE MODULAR ROBOT TIME PROTOCOL

MRTP works in two steps. The first step initializes the system: election of a central module as the time master *TM*, construction of a spanning tree and initialization of the global clock. In the second step, the time master periodically synchronizes the slave modules.

4.5.1/ METHOD TO COMPENSATE FOR COMMUNICATION DELAYS

The method of compensating for communication delays in MRTP has to be carefully selected depending on the target system. The choice of this method has a direct impact on both the precision of the synchronization and its efficiency in terms of communications. The precision of an approach mainly depends on the hardware-clock precision, its resolution and the communication mechanism. In Section 4.7.1.3, we describe a procedure to experimentally evaluate the precision of a given approach over multiple hops.

In that section, we also show that, in our target system, i.e., the Blinky Blocks, PRED is on average more precise than the other two existing methods that can be applied to this system (i.e., FD and RTT). Moreover, PRED uses a unidirectional message exchange while RTT requires a bidirectional message exchange, thus incurring a larger communication overhead.

In the rest of this section, we describe MRTP, assuming PRED is used. Note that in practice, any method compatible with the target system can be used in MRTP.

4.5.2/ STEP 1: INITIALIZATION

Time Master Election A module is elected as the time master using an external algorithm. Different criteria can be used for the election of the time master (e.g., minimum-identifier node, etc.). Modular robotic systems with neighbor-to-neighbor communications form large-diameter networks (see Section 2.2.5).

To achieve a better synchronization precision, we recommend electing a central module as the time master, i.e., a node that tends to minimize the maximum or the average hop distance to any other module. Placing the time master close to the center of the system reduces the time of the synchronization phases and increases the overall precision because cumulative estimations are made at every hop. Note that we do not claim that one can infer the synchronization precision of MRTP knowing the diameter of the target network. We only suggest a suitable position for the time master in a given system. Of course, the node density, the traffic distribution and the clock distribution may have an

impact on the overall synchronization precision.

Any center election algorithm can be used to elect a central module. We suggest using one of the algorithms defined in the previous chapter (k-BFS SumSweep, ABC-Center or PC2LE) or the algorithm presented in [Kim et al., 2013]. These algorithms scale well in terms of memory usage and execution time.

To handle dynamic topology changes, a module launches a time master re-election if it detects a new neighbor or a neighbor departure, and the system goes through the whole initialization process again.

Breadth-First Spanning Tree Construction At the end of the election process, our protocol creates a breadth-first spanning tree rooted at the time master. The CHEUNG-BFS-ST-CB algorithm, presented in Section 3.5.1, can be used. This algorithm guarantees that modules at distance d_{TM} hops of the time master in the physical configuration, are at distance d_{TM} hops in the tree. Logical neighbors in the tree are neighbors in the physical configuration. At this point, every module knows its parent and children in the tree. This tree will be used to recursively propagate synchronization waves from the time master through the system. As explained in Section 4.3.2, this approach is, in compact systems running under our assumptions, more communication-efficient than infrastructure-less network-wide flooding based approaches.

Global Clock Initialization Initially, slave modules estimate the global time with their local time. Slave modules adjust their estimation of the global time during synchronization phases, in the second step of MRTP. When a new time master is elected, modules keep their previous estimation of the global time but do not keep the previous corrections of the clock skew. They can indeed disturb the synchronization process when two distinct systems are merged together.

Since time cannot run backward, clocks ahead of the global timescale have to slow down or to wait during the synchronization process, and clocks behind the global timescale have to jump to it. To make time synchronization faster, the global time, held by the time master, is initially set to an estimation of the most advanced global time in the system using the convergecast-max-time algorithm. Note that this approach can cause important jumps into the future.

The pseudo-code of convergecast-max-time for any module M_i is provided in Algorithm 12. At any time, a module M_i estimates the maximum global time with:

$$Y^{M_i}(t) = L^{M_i}(t) + offset^{M_i}(t) \tag{4.4}$$

with $offset^{M_i}(t)$ being the estimated offset between the estimation of M_i concerning the maximum global time in the system and the local clock of M_i at time t. Initially, M_i considers it has the maximum global time (line 2). This algorithm uses a single type of message, namely *BACK* message. Every *BACK* message m is timestamped twice at the data-link layer: the sender M_j inserts $Y^{M_j}(t_s^m)$ just before transmission starts and the receiver M_k in-

Input : M_p // parent in the tree
$Children$ // set of children in the tree

1 **Initialization** of M_i at time t_{init}:
2 $offset \leftarrow G^{M_i}(t_{init}) - L^{M_i}(t_{init})$; $Wait \leftarrow Children$;
3 **if** $M_p = \perp$ **then**
4 // convergecast-max-time terminates
5 **else if** $Wait = \emptyset$ **then**
6 send m = BACK(_,_) to M_p;
// M_p will receive BACK($Y^{M_i}(t_s^m) = L^{M_i}(t_s^m) + offset^{M_i}(t_s^m), L^{M_p}(t_r^m)$) at the application layer. $Y^{M_i}(t_s^m)$ is inserted by M_i at the data-link layer, just before transmission start. M_p will insert $L^{M_p}(t_r^m)$ upon reception, at the data-link layer.

7 **When** m = BACK($Y^{M_c}(t_s^m), L^{M_i}(t_r^m)$) **is received by** M_i **from** M_c such that $M_c \in Children$ **do**:
8 $Y^{M_c}(t_r^m) \leftarrow Y^{M_c}(t_s^m) + T^m_{transfer}$;
9 $offset \leftarrow max(offset, Y^{M_c}(t_r^m) - L^{M_i}(t_r^m))$;
10 $Wait \leftarrow Wait - \{M_c\}$;
11 **if** $M_p = \perp$ **then**
12 // convergecast-max-time terminates
13 **else if** $Wait = \emptyset$ **then**
14 send m' = BACK(_, _) to M_p;
// As explain in comment line 6, M_p will receive BACK($Y^{M_i}(t_s^{m'}), L^{M_p}(t_r^{m'})$) at the application layer.

Algorithm 12: The convergecast-max-time algorithm for a module, M_i.

serts $L^{M_k}(t_r^m)$ upon complete reception (see Figure 4.3).Each leaf module sends a *BACK* message to its parent (line 6). Every non-leaf module waits for a *BACK* message from all its children. When M_i receives a *BACK*$(Y^{M_c}(t_s^m), L^{M_i}(t_r^m))$ message m from one of its children M_c, M_i estimates $Y^{M_c}(t_r^m) \approx Y^{M_c}(t_s^m) + T^m_{transfer}$ using the PRED method (line 8) and adjusts $offset^{M_i}(t)$ accordingly (line 9). When M_i has received a *BACK* message from all its children, it sends in turn a *BACK* message to its parent. When the convergecast terminates (lines 4 or 12), the time master has an estimation of the maximum global time in the system $Y^{TM}(t)$. The time master then sets the global timescale $G(t)$ to $Y^{TM}(t)$. The convergecast-max-time algorithm neglects the effect of clock skew, and considers offsets to be constant in the system during convergecast.

4.5.3/ Step 2: Periodic Synchronization

The time master holds the global timescale and periodically initiates synchronization phases. During each synchronization phase, the time master disseminates the current global time along the edges of the spanning tree built in the first step. $\tilde{G}(t)$, an estimation of the global time, is disseminated through the spanning tree, module-by-module, starting from the time master. At each hop, the transmitted time is updated to take into account communication delays and time of residence in intermediate modules. Slave modules use a linear model to compensate for clock skew. As explained in the related-work section, this is a common choice.

The time master starts a synchronization phase by sending the actual global time to all

its children. Algorithm 13 details the synchronization process of any slave module M_i.

Input : M_p // parent in the tree
$Children$ // set of children in the tree
w // maximum number of synchronization points used for linear regressions

1 **Initialization** of M_i:
2 $a \leftarrow 1.0$; $b \leftarrow 0$; $W \leftarrow \emptyset$;

3 **When** m = SYNC($\tilde{G}(t_s^m)$,$L^{M_i}(t_r^m)$) **is received by** M_i **from** its parent M_p **do**:
4 $\tilde{G}(t_r^m) = \tilde{G}(t_s^m) + T^m_{transfer}$;
5 **if** $|W| = w$ **then**
6 $W \leftarrow W - \{\underset{\tilde{G}(t)}{\text{argmin}}\ W(<\tilde{G}(t), L(t)>)\}$;
7 $W \leftarrow W \cup <\tilde{G}(t_r^m), L^{M_i}(t_r^m)>$;
8 computeLinearRegression(a, b, W);
9 **for each** $M_c \in Children$ **do**
10 send m' = SYNC(_, _) to M_c;
// M_c will receive SYNC($\tilde{G}(t_s^{m'}), L^{M_c}(t_r^{m'})$) at the application layer.
$\tilde{G}(t_s^{m'}) = \tilde{G}(t_r^m) + a^{M_i}(W^{M_i}(t_s^{m'})) * (L^{M_i}(t_s^{m'}) - L^{M_i}(t_r^m))$ is inserted at the data-link layer, just before transmission start. M_c will insert $L^{M_c}(t_r^{m'})$ upon reception, at the data-link layer.

Algorithm 13: Synchronization protocol for a slave module, M_i.

Time-stamping and Global Time Estimation The synchronization process uses a single type of message *SYNC*. Every *SYNC* message m is timestamped twice at the data-link layer: the sender, M_j, inserts $\tilde{G}(t_s^m)$ just before transmission starts and the receiver, M_k, inserts $L^{M_k}(t_r^m)$ upon complete reception. When M_i receives a *SYNC*$(\tilde{G}(t_s^m), L^{M_i}(t_r^m))$ message m from its parent, M_i computes $\tilde{G}(t_r^m) = \tilde{G}(t_s^m) + T^m_{transfer}$, an estimation of the global time at the reception of the synchronization message, using the PRED method (line 4). $<\tilde{G}(t_r^m), L^{M_i}(t_r^m)>$ forms a synchronization point that contains both the local clock value of M_i and the estimation of the global time at nearly the same real time. M_i can estimate its relative synchronization error with respect to the global time using Equation (4.5).

$$\tilde{\epsilon}^{M_i}(t) = G^{M_i}(t_r^m) - \tilde{G}(t_r^m) \tag{4.5}$$

Global Clock Adjustment M_i computes $a^{M_i}(W^{M_i}(t))$ and $b^{M_i}(W^{M_i}(t))$ such that

$$\tilde{G}(t) \sim a^{M_i}(W^{M_i}(t)) \times L^{M_i}(t) + b^{M_i}(W^{M_i}(t)) \tag{4.6}$$

using least-squares linear regression based on $W^{M_i}(t)$, a window of the last w synchronization points (line 8). $a^{M_i}(W^{M_i}(t))$ denotes the M_i estimated skew with respect to the global time, and $b^{M_i}(W^{M_i}(t))$ its estimated offset at time t. This mechanism compensates for clock skew and enables modules to be synchronized less frequently without degrading the synchronization precision. In order to preserve time monotonicity, our protocol

prevents $G^{M_i}(t)$ from running backward:

$$\forall (t,t'), t \geq t',\ G^{M_i}(t) = \max\left(G^{M_i}(t'),\ a^{M_i}\left(W^{M_i}(t)\right) \times L^{M_i}(t) + b^{M_i}\left(W^{M_i}(t)\right)\right) \tag{4.7}$$

If a new computed model leads to an estimated global time behind the maximum time already reached by $G^{M_i}(t)$, then $G^{M_i}(t)$ is blocked until the new model reaches this maximum time. Otherwise, $G^{M_i}(t)$ jumps into the future.

Global Time Dissemination M_i then sends a *SYNC* message m' to each of its children M_c in the tree (line 10). At the data-link layer, M_i inserts

$$\tilde{G}(t_s^{m'}) = \tilde{G}(t_r^m) + a^{M_i}\left(W^{M_i}\left(t_s^{m'}\right)\right) \times \left(L^{M_i}\left(t_s^{m'}\right) - L^{M_i}\left(t_r^m\right)\right) \tag{4.8}$$

into m', just before it starts to transmit the frame over the communication medium. This compensates for the time of residence at module M_i, assuming the M_i clock skew to be constant and equal to $a^{M_i}(W^{M_i}(t_s^{m'}))$ during this time. M_c inserts its local time $L^{M_c}(t_r^{m'})$ into the incoming message at the data-link layer, immediately after M_c has pulled the synchronization message from the interface buffer. At the M_c application layer, m' contains $\tilde{G}(t_s^{m'})$ and $L^{M_c}(t_r^{m'})$. M_c then repeats the same synchronization process as M_i.

Synchronization Periods Our protocol contains two synchronization phases: a calibration phase and a runtime phase. During the calibration phase, modules are more frequently synchronized with a period P_{ca} in order to collect enough synchronization points to compute skew models while preserving a satisfying level of precision. The calibration phase lasts $w \times P_{ca}$. Then, during the runtime phase, modules are synchronized less frequently, with a period P_{ru}, and use the computed models to compensate for clock skew. The values of w, P_{ca}, and P_{ru} have to be chosen according to the target platform hardware and the desired precision, with resource usage in mind.

In our experimental evaluation, we empirically selected $w = 5$, $P_{ca} = 2\ seconds$ and $P_{ru} = 5\ seconds$ (unless otherwise mentioned). These values provide, in our target platform, a satisfactory precision at a reasonable cost in terms of communications and computations.

4.6/ THE TARGET SYSTEM: THE BLINKY BLOCKS

We implemented MRTP and evaluated it on the Blinky Blocks system using both hardware prototypes and simulations on VisibleSim. Figure 4.4 shows MRTP running on hardware Blinky Blocks. This section presents the characteristics of the Blinky Blocks local clocks and communication systems on the hardware prototypes along with the simulation models used in the simulations.

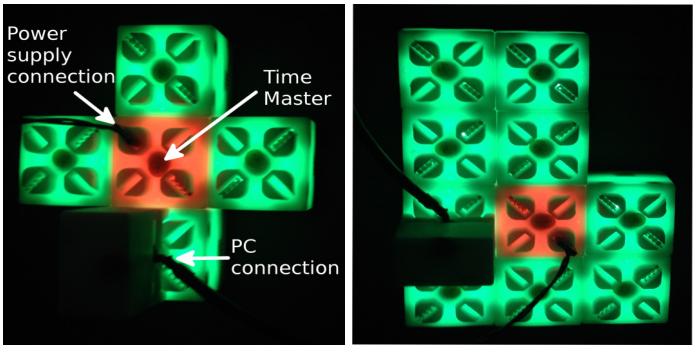


Figure 4.4: Two Blinky Blocks systems synchronized using MRTP. On the left, the system forms a cross. On the right, blocks are deployed in a doubled L-configuration. In both configurations, the time master, in red, is connected to the power supply. Slave modules are in green. Experimental data are sent by the systems to the PC through a serial cable.

4.6.1/ Local Clock Properties

Hardware System Each module maintains its local time using a Real-Time Counter (RTC) driven by an internal RC oscillator running at a frequency of 1.024 kHz with an accuracy of 1% (10,000 ppm), at 3V and 25℃ [ATMEL, 2016]. The RTC counts the time elapsed since the module started with a resolution of about 0.98 millisecond[5]. Thus, the synchronization precision results announced in the evaluation section are actually expressed in 0.98 a millisecond, even though we express them in milliseconds for the sake of simplicity. It is important to understand that these oscillators exhibit a very poor accuracy and low resolution that directly affects the performance of our protocol. For instance, a frequency deviation of 1% causes a clock error of approximately 10 milliseconds per second. Most previous work on time synchronization, e.g., [Elson et al., 2002, Ganeriwal et al., 2003, Maróti et al., 2004, Schenato et al., 2011], was evaluated on devices equipped with crystal oscillators that have a typical accuracy between 0.0001% and 0.01% (1 to 100 ppm) and a resolution in the order of tens of microseconds. Under constant temperature and constant supply voltage conditions, RC oscillators are fairly stable over a short period of time. As shown in Figure 4.5, Blinky Blocks local clocks tend to drift apart in a roughly linear fashion in the short term.

Simulation Model In [Allan, 1987], the authors propose a general model for oscillators:

$$L^{M_i}(t) = \frac{1}{2}D^{M_i}t^2 + y_0^{M_i}t + x_0^{M_i} + \eta^{M_i}(t) \tag{4.9}$$

where t is the real time (i.e., simulation time), $L(t)$ is the local time, x_0 is the time offset, y_0 is the frequency offset, D is the frequency drift and $\eta(t)$ is a random noise. As explained in [Allan, 1987], y_0 and D may vary over time (e.g., due to aging, temperature variations,

[5] Resolution= $\frac{1}{1.024} \approx 0.98ms$

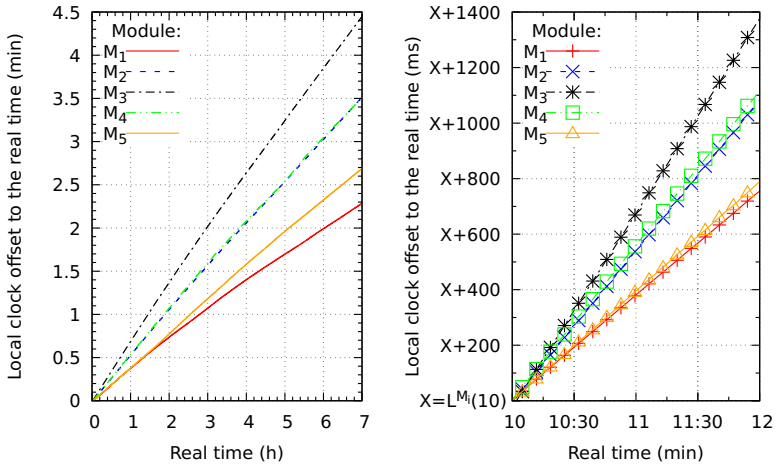


Figure 4.5: Local clock offset with respect to the real time $(L^{M_i}(t) - t)$. The plot on the left shows the long-term deviation of the local clocks, while the plot on the right shows these deviations in the shorter term. The PRED method was used to compensate for communication delays.

etc.). For the sake of simplicity, we consider them to be constant and express their small variations in the noise signal $\eta(t)$.

We assume that Blinky Blocks clocks follow the model shown in (4.9). We conducted experiments on hardware using Blinky Blocks in order to compute model parameters. We used a system of five blocks deployed in a cross configuration (see Figure 4.4) to collect time reference points $< t, L^{M_i}(t) >$, with i being the block unique identifier, every 10 seconds during 7 hours (see Figure 4.5). The real time t was provided by a computer. We assumed the computer clock to be perfect. We use the PRED method of compensating for communication delays.

Figure 4.6 shows the distribution of the parameter values obtained using polynomial regression with R. The parameters D and y_0 seem normally distributed. As a consequence, we randomly generate clock parameters following normal distributions with the corresponding mean and standard deviation (see Table 4.2). Noise signals are the residual standard errors. We extract the 5 noise signals and replay them in our simulations.

Parameters	Simulation Model
D (μs^{-1})	$\mathcal{N}(7.132315 \times 10^{-14}, 5.349995 \times 10^{-14})$
y_0 $(none)$	$\mathcal{N}(0.9911011, 0.002114563)$
x_0 (μs)	additive inverse of the simulation time at module start-up
η (μs)	Noise replayed from extracted data signals

Table 4.2: Blinky Blocks hardware-clock model parameters used in VisibleSim. $\mathcal{N}(\mu, \sigma)$ refers to the normal probabilistic law, with μ being the mean and σ the standard deviation.

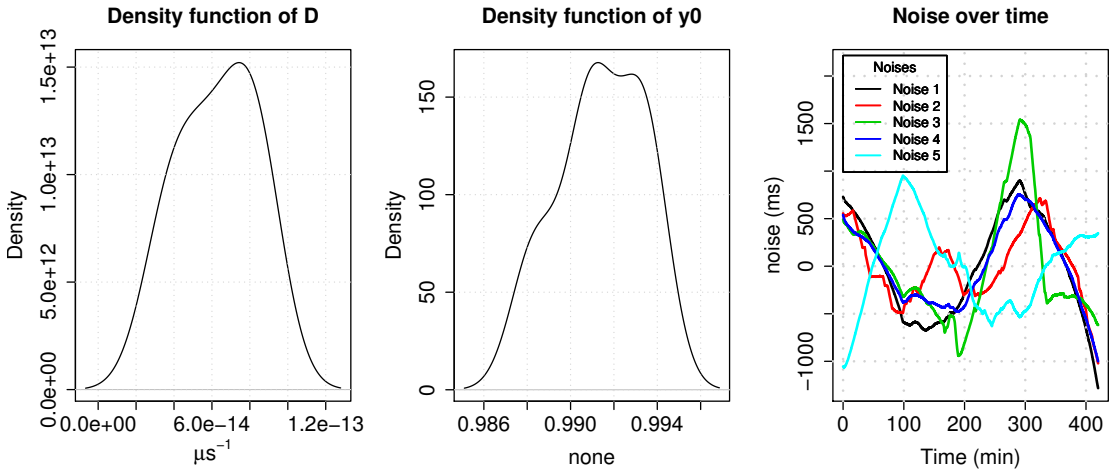


Figure 4.6: Statistics on the parameters of the model used to simulate clocks. From left to right: D density function, $y0$ density function and the noise signals over the time.

4.6.2/ COMMUNICATION PROPERTIES

Hardware System We recall that Blinky Blocks use full-duplex neighbor-to-neighbor communications over serial links controlled by Universal Asynchronous Receivers/Transmitters (UARTs) configured with a bitrate of 38.4 kBauds. Modules exchange messages that contain up to 17 bytes of application data. A message is sent over the link into a frame composed of a minimum of 21 bytes: 17 bytes of payload data, 2 bytes for data related to message handling (active messaging [Eicken et al., 1992]) and 2 bytes of control (i.e., a frame delimiter byte and a checksum byte). Some special bytes need to be escaped using an extra byte in order to dissociate command bytes from data ones. Thus, the number of bytes actually sent on the link varies a little according to the data being sent.

A frame is transferred byte per byte to/from the UART. The transfer is interrupt-controlled, i.e., the UART generates an interrupt when it has finished transmitting or receiving a byte. The transmission time starts when the first byte of data is moved to the UART buffer and ends when the last byte leaves this buffer. The reception time starts when the first byte of data is received by the UART and ends when the last byte is received.

Transfer Time Estimation The PRED method used to compensate for communication delays assumes that the transfer time, defined in subsection 4.4.2, is predictable. The transfer time includes the transmission time, the propagation time and the reception time. The Blinky Blocks are identical and physically connected, thus the propagation time between two neighbor modules can be considered to be deterministic. The transmission time and the reception time of a message depend on the actual frame size and on the communication rate.

$T_{transfer}$ can be estimated using two-way timestamped-message exchanges (see Fig-

ures 4.7-4.8 and Equation (4.10)). Equation (4.10) assumes the communication delays for frames of same size to be symmetrical. In addition, the exchange of messages is assumed to be fast enough so that the skew between the clock of the two modules is insignificant during the exchange.

$$T_{transfer} \approx \frac{(L^{M_2}(t_r^{m'}) - L^{M_2}(t_s^{m})) - (L^{M_1}(t_s^{m'}) - L^{M_1}(t_r^{m}))}{2} \tag{4.10}$$

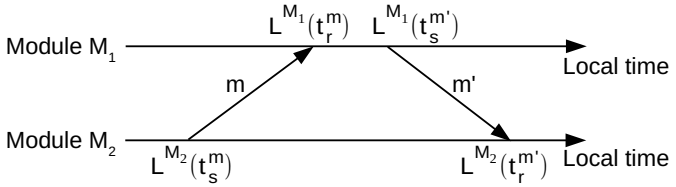


Figure 4.7: Scheme of a two-way message exchange between two blocks.

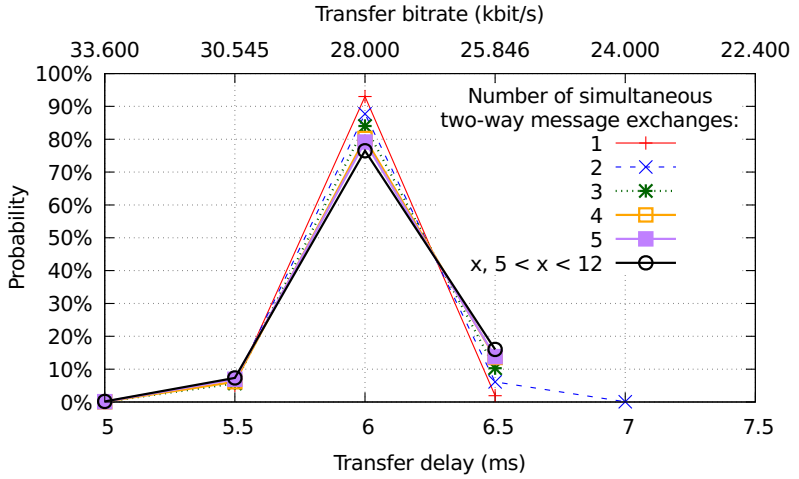


Figure 4.8: Transfer delay/rate distribution of 21-byte-long frames.

We experimentally measured $\widetilde{T}_{transfer}$ for 300,000 two-way message exchanges between neighbor modules in sparse and compact Blinky Blocks systems (see Figure 4.8). We observed that $\widetilde{T}_{transfer}$ is always between 5 and 7 milliseconds. On average, $\widetilde{T}_{transfer}$ of 21-byte long frames varies slightly around 6 milliseconds, depending on the number of simultaneous communications. Moreover, at the resolution of 1 millisecond, the transfer time of identical-length frames is fairly constant. A transfer time of 6 milliseconds for a 21-byte long frame corresponds to a transfer rate of 28 kbit/s. Based on these results, we consider that the transfer rate of a message can be estimated by $\widetilde{R}_{transfer} = 28\ kbit/s$. As a consequence, we use Equation (4.11) to estimate the transfer delay of a message and to compensate for communication delays in the PRED method.

$$\widetilde{T}_{transfer} = \frac{\text{frame size}}{\widetilde{R}_{transfer}} \tag{4.11}$$

Simulation Model In order to accurately simulate the time, our simulation model takes into account the timeout triggering time, the processing time, the queueing delays, and the transfer rate of the messages (see Figure 4.9). We did not observe any node crash or any transmission failure or message loss during the experiments of the previous subsection, when the network is not overwhelmed. Thus, our simulation model does not incorporate any special mechanism to mimic such phenomena. Table 4.3 summarizes the different random variables of our model.

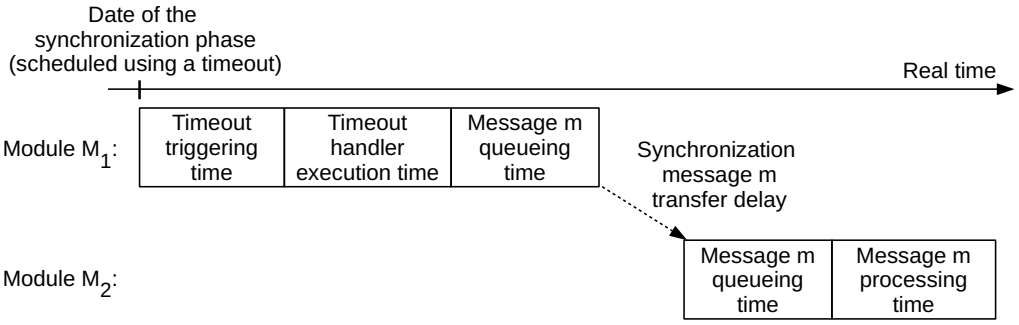


Figure 4.9: Workflow of the communication model used for the simulation of time synchronization protocols. In this example, module M_1 has scheduled a synchronization phase. Upon timeout expiration, module M_1 executes the synchronization procedure and sends a synchronization message to module M_2 which will process it after a possible delay due to queueing.

Timeout triggering time The timeout triggering time is the amount of time a module needs to trigger an action scheduled using a software timeout (e.g., the synchronization timeout that initiates a synchronization phase). In the Blinky Blocks firmware, software timeouts are checked with a frequency of 2000 Hz. Thus, if we neglect the interrupt delay, an action scheduled at time t can be executed at any time t', such that $t \leq t' \leq t + 500\mu s$.

Processing time We use the micro-controller clock running at 32 MHz (nanosecond scale resolution) to measure the processing time of the synchronization-timeout handler and the synchronization-message handler. We define two generic models to simulate the message-handler processing time: one for handlers with low-computation cost (e.g., clock adjustment without linear regression) and one for handlers with medium-computation cost (e.g., clock adjustment with linear regression computation on a window of 5 measures).

Note that the queueing and transfer delays include some processing time. In our evaluation, modules were running a rather simple application, in which every module periodically changes its color based on the current global time and does nothing the rest of the time using an active sleep (while loop with a time limit). Thus, they were actually computing all of the time. The transfer time includes the interrupt time to fetch bytes from the interface buffer. Our target platform (and many others) uses interrupt-driven communications. Hence, only a very few elementary micro-controller instructions are executed before a byte is fetched. We reasonably assume that interrupts are never disabled and that there

are not a large number of interrupts to be simultaneously handled. The queueing delays include interrupt time to enter the routine that handles incoming messages and the time to handle potential messages that were already present in the queue at the message arrival.

Queueing delays and network load VisibleSim uses a queueing system to handle both incoming and outgoing messages. We propose two queue load models. The first model is dedicated to lightly loaded networks where modules only exchange neighborhood management messages, with a period of 500 milliseconds. The second model is intended to simulate moderate network traffic due to extra-applications running on the nodes. In this model, in addition to simulating the neighborhood management messages, the queue occupancy at a message arrival follows a Poisson distribution of mean 1. This simulates a moderate network traffic in which message queues contain, most of the time, 0 to 2 messages and in a few cases more messages. The light-load model is used in the experiments of subsection 4.7.1. The moderate-load model is used in our evaluation on large-scale systems (see Section 4.7.2).

Transfer rate Below the millisecond unit, the transfer rate is scenario-dependent. It depends, for instance, on the number of simultaneous communications. For each experiment performed on the hardware platform, we empirically derive the average system transfer rate using statistics on the round-trip time. We use similar experiments to the ones presented in subsection 4.7.1.3. We define three transfer rate models, namely for sparse, intermediate and compact systems. In a given simulation, all the modules use the same transfer rate model. The model for sparse systems is used in the experiments of subsection 4.7.1.3, on the line system. The model for intermediate systems is used in the experiments of subsections 4.7.1.5 and 4.7.1.6, on the L-shaped system (see Figure 4.4). The model for compact systems is used in our evaluation on large-scale systems (see Section 4.7.2).

4.7/ EXPERIMENTAL EVALUATION

This section presents our experimental evaluation of MRTP, performed both on hardware Blinky Blocks and in the VisibleSim simulator. Through our experiments, we show the effectiveness, the efficiency and the scalability of our protocol. More precisely, we first evaluate the precision of MRTP on hardware and show through some examples that VisibleSim accurately simulates Blinky Blocks systems. Then, we use VisibleSim to evaluate the performance of MRTP in large-scale systems and to compare it to existing synchronization protocols in terms of precision, time of convergence and communication efficiency. Unless otherwise mentioned, we use the PRED method to compensate for communication delays in MRTP.

Parameters			Value
Timeouts	Triggering time (s)		$\mathcal{U}(0, 500 \times 10^{-6})$
	Processing time (s)		$\mathcal{U}(250 \times 10^{-6}, 300 \times 10^{-6})$
Messages	Queue occupancy at arrival	Light load	neighborhood management
		Moderate load	neighborhood management + $\mathcal{P}(1)$
	Transfer rate $(kbit/s)$	Sparse systems (e.g., line system)	$\mathcal{N}(28.134, 0.660)$
		Intermediate systems (e.g., L-shaped systems)	$\mathcal{N}(28.085, 0.938)$
		Compact systems (e.g., ball systems)	$\mathcal{N}(27.696, 1.143)$
	Processing time (s)	Low complexity	$\mathcal{U}(250 \times 10^{-6}, 300 \times 10^{-6})$
		Medium complexity	$\mathcal{U}(475 \times 10^{-6}, 525 \times 10^{-6})$

Table 4.3: Communication model used for the evaluation of time synchronization protocols. $\mathcal{N}(\mu, \sigma)$ refers to the normal probabilistic law, with μ being the mean and σ, the standard deviation. $\mathcal{U}(l, u)$ refers to the uniform probabilistic law with the minimum value l and the maximum value u. $\mathcal{P}(\lambda)$ refers to the Poisson probabilistic law with λ mean.

4.7.1/ Evaluation on Hardware and Validation of VisibleSim

In this subsection, we evaluate the precision of the synchronization achieved by MRTP on the Blinky Blocks hardware. In addition, we show that VisibleSim accurately simulates Blinky Blocks systems.

4.7.1.1/ Methodology

We first use color changes to show that MRTP can potentially manage systems composed of up to 27,775 Blinky Blocks.

Then, we show how the hop distance impacts the precision of the estimated global time $\tilde{G}(t)$ disseminated through the network during synchronization phases. We compare different methods to compensate for communication delays and show that the PRED method is on average the most accurate in our target platform. Furthermore, we show that within a few hops, $\tilde{G}(t)$ can be used as a reference time to estimate the relative synchronization error of the Blinky Blocks with respect to the global time. The relative synchronization error can thus be estimated using Equation (4.5).

We then use this estimation to study the local clock behaviors and to show the impact of various parameters on the precision of our protocol.

All experiments presented in this subsection were one-hour long. Unless otherwise mentioned, modules were synchronized every 2 seconds in the calibration phase, then every 5 seconds in the runtime phase and modules used five synchronization points for the linear regressions. These values were empirically chosen with the aim of obtaining, a satisfactory synchronization precision in practice, at reasonable computation and communication costs.

4.7.1.2/ EVALUATION OF THE PRECISION OF MRTP USING COLOR CHANGES

Measuring clock offsets using message exchanges is as challenging as performing time synchronization because time keeps going during communications.

In this subsection, we apply MRTP to a system of 28 Blinky Blocks[6] that have to simultaneously change their color. Potential delays between module color changes reflect the synchronization error of the modules. Modules are connected in a line topology. The time master is manually placed at an extremity of the system and it synchronizes the other modules every 500 milliseconds. With a such runtime synchronization period, every link of the synchronization tree is theoretically used by MRTP only about 1.2% of the time[7]. Slave modules have to simultaneously change their color every 3 seconds. This experiment was recorded using a 40-millisecond-resolution camera.

We observed that every time the system started to change its color, all slave modules changed their color in the next image, 40 milliseconds later (see Figure 4.10). Hence, MRTP is potentially able to synchronize a system with a radius of up to 27 hops to a less than 40 milliseconds, if the time master is at the center of that system. To give an order of magnitude, a Blinky Blocks system with a radius of 27 hops can be composed of up to 27,775 modules and have a diameter of 54 hops (using formulas demonstrated in Section A.5.2).

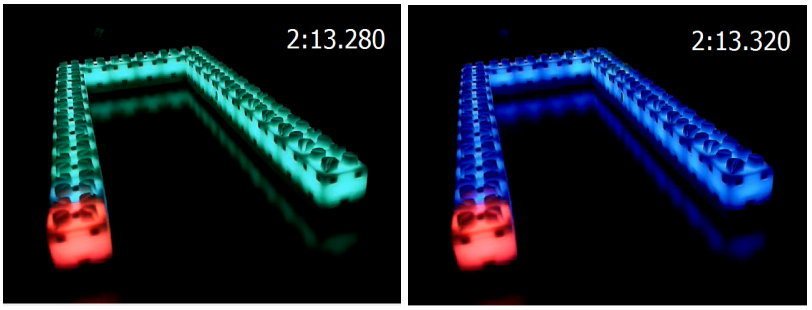


Figure 4.10: Two successive images of a video recording 28 Blinky Blocks connected in a line topology. The time master is in red. Slave modules have to simultaneously change their color every 3 seconds. On the left, a color change starts in the system. On the right, 40 milliseconds later, the color of every slave module has changed.

In the next subsections, we present a more precise and automated evaluation of MRTP.

[6] At the time of the evaluation of MRTP, we only had at our disposal 28 hardware Blinky Blocks.

[7] $\frac{T_{transfer}}{P_{ru}} \approx \frac{6}{500} \approx 1.2\%$ (without retransmission due to potential message loss or corruption)

4.7.1.3/ Impact of the Hop Distance and the Method to Compensate for Communication Delays on the Precision of the Disseminated Global Time $\tilde{G}(t)$

We expect that the estimation of the global time, $\tilde{G}(t)$, disseminated during the synchronization phases gets less precise as the depth of the synchronization tree increases because small but cumulative errors in the estimation of the global time are made at every hop. In this section, we first propose a generic method to evaluate compensation delay methods over multiple hops. Then, we present results obtained using the FD, RTT and PRED methods of compensating for communication delays (see Sections 4.3.3 and 4.4.3). We show that the PRED method is on average more accurate. Finally, we show that within a few hops, $\tilde{G}(t)$ can be used as a reference time to estimate the synchronization error of the Blinky Blocks.

Methodology We evaluate the precision of $\tilde{G}(t)$ using virtual modules emulated on Blinky Blocks hardware systems. Figure 4.11 gives the intuition behind our experiments in a line system. This method, inspired by the approach presented in [Römer et al., 2005], allows us to compare the estimated global time received by the module M_{2n-1} to the actual global time held by the time master $TM = M_1$, because these two modules are emulated on the same physical block and can both read the actual global time $G(t)$.

In the example depicted in Figure 4.11, every physical block hosts 2 virtual modules, except for one block. Each slave virtual module maintains its own estimation of the global time. The synchronization tree rooted at the time master *TM* links the virtual modules together in a virtual line, so that neighbor modules in the tree are hosted on a separate physical block. The leaf module M_{2n-1} is at a distance of $2(n-1)$ hops from *TM* in the synchronization tree. M_{2n-1} computes the global-time dissemination error as $G(t) - \tilde{G}(t)$.

In our experiments, we generalize the example of the virtual line to measure the global-time dissemination error versus the hop distance in arbitrary systems. Modules host a number of virtual modules equal to the diameter of the system, and every physical module initiates a return trip to the root of the tree. The root of the tree receives timing messages that have physically traveled from 2 hops to $2(d-1)$ hops (or $2d-1$, if the diameter is odd).

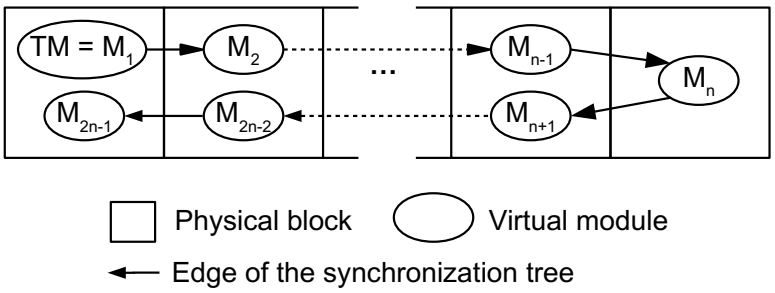


Figure 4.11: Scheme of a virtual line of emulated modules on hardware Blinky Blocks connected in a line.

Results Figure 4.12 and Table 4.4 show the impact of the hop distance on the global-time dissemination error. As expected, the precision of the disseminated global time decreases with the hop distance. As stated in Section 4.3.5, the absolute mean error increases linearly with the number of hops and the standard error tends to increase with the square root of the number of hops. As a consequence, placing the time master at the center of the system appears to be a judicious choice.

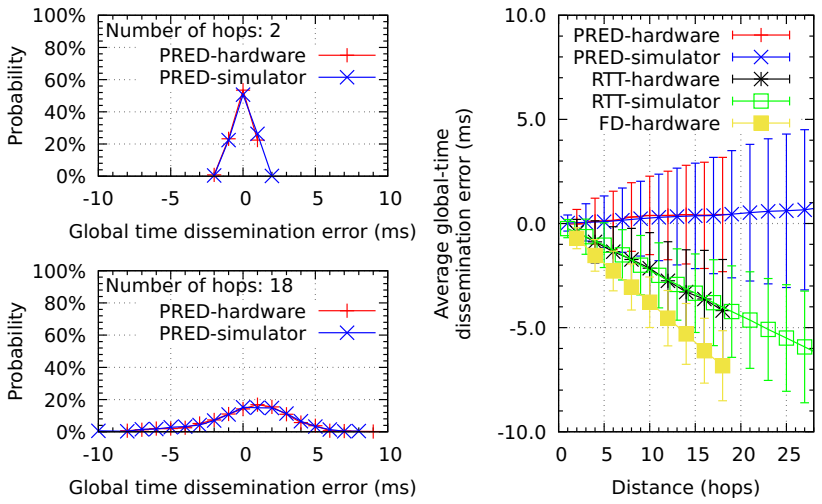


Figure 4.12: Global time dissemination error (± standard deviation) in MRTP according to the hop distance. On the left, the distribution of the error. On the right, the average error (± standard deviation).

Compensation delay method	Average global-time dissemination error (ms)			
	Line configuration		Compact configuration	
	2 hops	4 hops	2 hops	4 hops
PRED	-0.03 ± 0.70	-0.11 ± 1.11	-0.27 ± 0.67	-0.36 ± 1.02
RTT	-0.42 ± 0.62	-0.88 ± 1.01	-0.50 ± 0.63	-0.80 ± 0.97
FD	-0.71 ± 0.50	-1.53 ± 0.76	-0.87 ± 0.54	-1.63 ± 0.80

Table 4.4: Average dissemination error (± standard deviation) with respect to the global time in MRTP for 2 and 4 hops using different methods of compensating for communication delays in the line and the compact systems.

It appears that PRED is on average more precise than FD and RTT methods in sparse and more compact systems. We observe that regardless of the distance, the error distribution of PRED seems Gaussian and nearly centered around zero. This is mainly due to the fact that, in Blinky Blocks systems, the average transfer delay of a frame is almost a round number at the millisecond scale.

Note that PRED has a more important standard deviation than the other two methods. FD has the smallest standard deviation as only the transfer time of a single byte is involved

in the estimation of the global time whereas PRED and RTT use the transfer time of complete messages. In future work, it would be interesting to evaluate the performance of FD combined with a method that would compensate for the dissemination error only after several hops when this error has become greater than the resolution of the clock and can effectively be compensated for. From now, we only consider the PRED method for the evaluation on hardware.

For a distance of 4 hops, 95% of the error measures are between [-2;2] milliseconds and the average error is close to zero. Because of the poor accuracy of the Blinky Blocks hardware clocks, we expect synchronization error using our protocol to be greater than 1 to 2 milliseconds. Thus, within a few hops, $\tilde{G}(t)$ can be used as a reference time to estimate the synchronization error of the Blinky Blocks. Upon reception of a synchronization message, a module M_i estimates its relative synchronization error with respect to the global time by $\tilde{\epsilon}^{M_i}(t) = G^{M_i}(t) - \tilde{G}(t)$. We do not use virtual modules any more in the rest of the evaluation.

4.7.1.4/ Analysis of the Local Clock Behavior and Impact of the Hardware Clock Stability on the Synchronization Precision

We measured the clock values of five blocks deployed in the cross-configuration depicted in Figure 4.4. The slave modules, denoted M_6, M_7, M_8 and M_9, ran under the same conditions: they were all synchronized using the same parameters and they were all neighbors of the time master which was connected to the power supply. Note that the physical modules used in these experiments differ from the ones used in Section 4.6 to compute our simulation model. As shown in Figure 4.13, local clocks seem to drift apart from the global time in a roughly linear fashion. As indicated in Table 4.5, using linear models to compensate for clock skew significantly increases the synchronization precision.

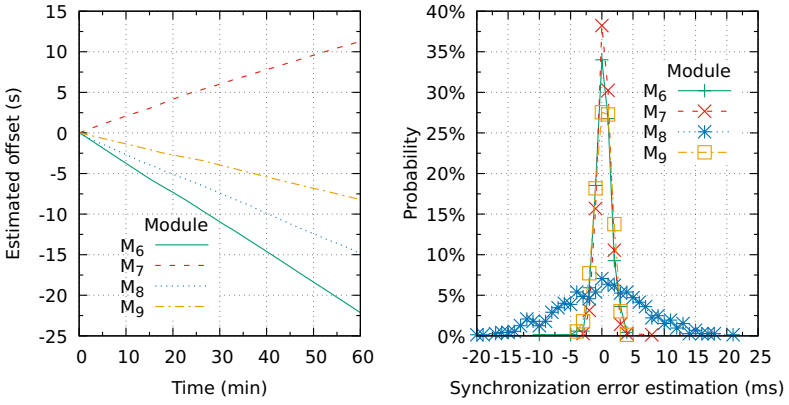


Figure 4.13: On the left, the estimated clock offset ($L^{M_i}(t) - \tilde{G}(t)$). On the right, the distribution of the relative synchronization error in MRTP.

The distributions of the estimated relative synchronization errors observed for all blocks are shown in Figure 4.13. The distributions seem to be Gaussian. They are bell-shaped

Clock skew compensation	Average (ms)	Standard deviation (ms)	Maximum absolute (ms)
Linear model	0.22	3.55	21
None	-12.13	18.05	67

Table 4.5: Statistics on the average relative synchronization error of the whole system showing the impact of using linear models to compensate for clock skew in MRTP.

and nearly centered around 0. All modules remain synchronized to a few milliseconds.

However, it must be noted that the distribution for M_8 is much more spread out than the others. It is shorter and flatter. Thus, M_8 is less precisely synchronized. Figure 4.14 shows the stability with respect to the global time and the synchronization error of the modules M_7 and M_8 during two minutes of the experiment. We observe that the synchronization error oscillates around 0 for both blocks. However, its magnitude is more important for block M_8 than for block M_7. This is because the local clocks of the two modules do not exhibit the same stability with respect to the global time. The offset with respect to the global time is less linear for the local clock of M_8, and its skew with respect to the global timescale varies much more.

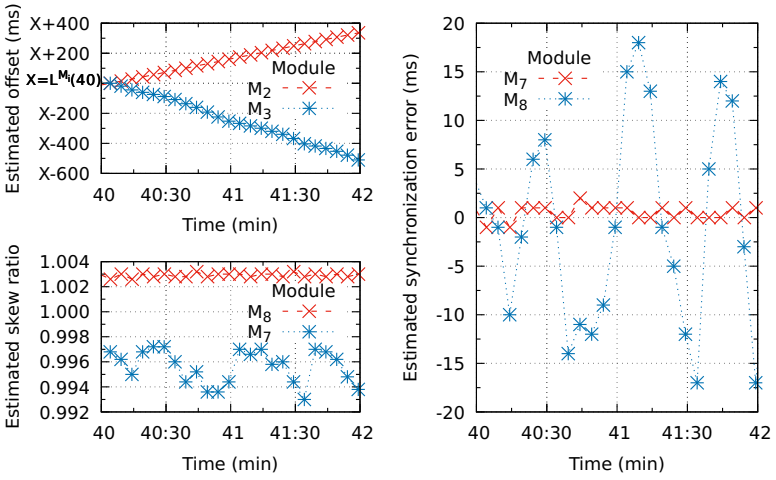


Figure 4.14: On the left, stability of the local clock of M_7 and M_8 with respect to the global timescale: above, the estimated offset $(L^{M_i}(40) - \tilde{G}(40)+(L^{M_i}(t)-\tilde{G}(t)))$ and below, the estimation of the estimated average skew ratio between synchronization points $(\frac{\Delta L^{M_i}(t)}{\Delta \tilde{G}(t)})$. On the right, the synchronization error of these two blocks.

Since all modules ran under the same experimental conditions, such an important difference in the synchronization precision should be due to the clock oscillator relative stability. Among the dozens of blocks we have, all modules behave similarly to M_6, M_7, and M_9, except for M_8. As a consequence, we consider M_8 to be an outlier and do not use it in the rest of the experiments. Note that we do not consider M_8 to compute our simulation model. We suggest that such outliers should be removed from the system when a precise time synchronization is required.

Furthermore, we experimentally checked that the hop distance to the block that is connected to the power supply has no significant impact on the individual synchronization precision.

4.7.1.5/ Impact of the Synchronization Periods on the Synchronization Precision

Figure 4.15 shows the impact of the synchronization periods on the relative synchronization error in the doubled L-shaped system depicted in Figure 4.4. Distributions seem to be Gaussian. They are all bell-shaped and centered around 0. For a runtime synchronization period of 5 seconds, the average relative synchronization error is equal to 0.22 millisecond.

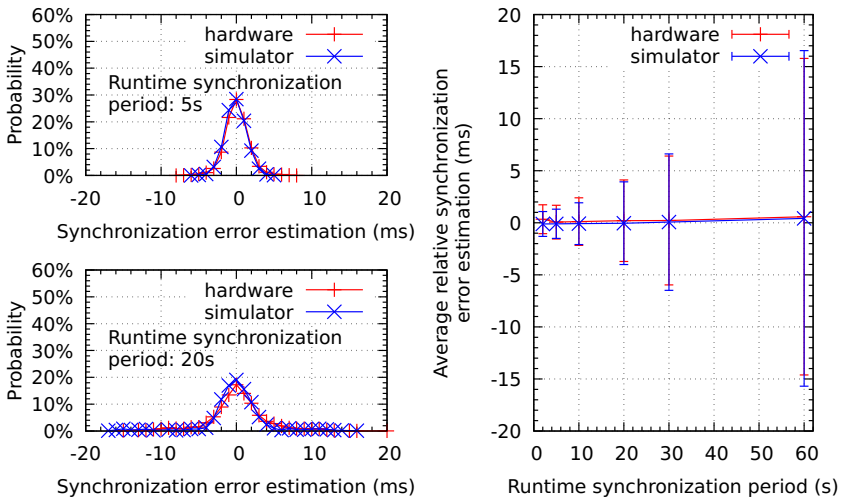


Figure 4.15: Relative synchronization error of the whole system as a function of the synchronization periods. On the left, the distribution of the error. On the right, the average error (± standard deviation).

We observe in Figure 4.15 that the distribution shape becomes shorter and larger, as the runtime synchronization period increases. The error dispersion reflects the synchronization error. The standard deviation increases with the runtime synchronization period. As a consequence, the longer the resynchronization interval is, the worse the synchronization precision will be. However, it must be noted that in all cases, the system stays synchronized to a few milliseconds. The average synchronization error amplitude remains below 4 milliseconds for runtime synchronization periods ranging from 2 seconds to 30 seconds. With a runtime period of 5 seconds, every link of the synchronization tree is theoretically used by MRTP only about 0.12% of the time during the runtime phase.

4.7.1.6/ Impact of the Number of Synchronization Points used for the Linear Regressions on the Synchronization Precision

Figure 4.16 shows the impact of the number of synchronization points used for the linear regressions on the synchronization error in the doubled L-shaped system depicted in Figure 4.4. With a running synchronization period of 5 seconds, we observe that the maximum synchronization precision is obtained using 5 synchronization points for the linear regressions. Indeed, when using 5 synchronization points, the relative synchronization error has a mean close to 0 and the smallest standard deviation. We suppose that the linear regression does not make it possible to properly capture the clock models when using less synchronization points. When using more than five synchronization points, the synchronization precision decreases as the window size increases. We believe, without proving it, that this is because the clock frequencies vary too quickly for a large number of observations.

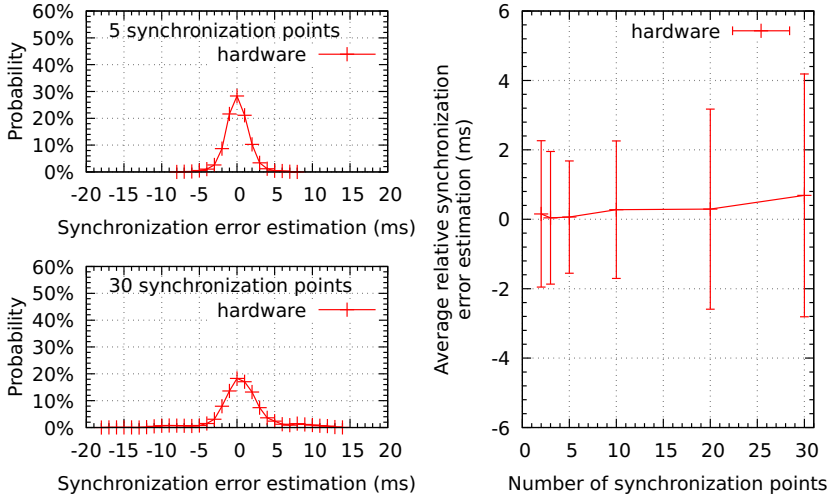


Figure 4.16: Relative synchronization error of the whole system as a function of the number of synchronization points used for the linear regressions. On the left, the distribution of the error. On the right, the average error (± standard deviation).

4.7.1.7/ Simulation Fidelity

As shown in figures 4.12 and 4.15, the results obtained using simulations closely match the results from the hardware-based experiments. Indeed, the results of the simulations have distributions and statistical measure values (i.e., means and standard deviations) that are almost identical to the results of the experiments on hardware systems. The global-time dissemination error according to the hop distance is well simulated even after many hops. Thus, we can safely assume that VisibleSim can be used to study the performance of synchronization protocols on large-scale Blinky Blocks systems.

Note that we do not simulate the experiment in Section 4.7.1.6 because we did not com-

pute the processing times for linear regression on a large number of synchronization points (i.e., more than five) in our simulation model.

4.7.2/ Large-Scale Evaluation and Comparison to Existing Protocols through Simulations

In this subsection, we use the VisibleSim simulator to evaluate the performance of our protocol and compare it with existing synchronization protocols. Section 4.7.2.1 presents the synchronization protocols and their variants that we consider for the comparisons.

We study the precision, the convergence time and the communication efficiency of the synchronization protocols on three systems of different sizes and diameters (see Table 4.6). These systems are organized in a ball topology, i.e., the largest network topology that can be formed for a given diameter (see Sections A.3 and A.5.2 for more details). We use this compact network topology because there is an increasing number of modules, therefore an increasing number of clock models, at a given network distance from any given module. Moreover, we consider the ball system composed of 27,775 modules to show that MRTP can effectively synchronize this system to a few milliseconds, as announced in subsection 4.7.1.2.

System	Size (modules)	Radius (hops)	Diameter (hops)
Ball(5)	231	5	10
Ball(15)	4,991	15	30
Ball(27)	27,775	27	54

Table 4.6: Network characteristics of the systems used for the evaluation of time synchronization protocols.

To compare protocols fairly, we evaluate them on identical systems, i.e., for all experiments, a module always has the same position, the same communication model and the same clock parameters. In addition, for centralized protocols, the time master always has the same communication model and clock parameters. Furthermore, the minimum-identifier module is deliberately placed at the extremity of the systems in order to show the impact of the maximum hop distance to the time master on the overall synchronization precision of the system.

All the experiments last for two hours. During the first hour, the system is left unsynchronized. Then, the modules start running one of the considered synchronization protocol. For all the protocols, we use a synchronization period of 5 seconds. In protocols that use a linear model to compensate for clock skew, modules perform the model parameter estimations using the last 5 synchronization points, unless otherwise mentioned. To evaluate the synchronization precision, we measure the maximum pairwise synchronization error every 3 seconds.

4.7.2.1/ Compared Synchronization Protocols and Modifications

We compare MRTP to leading protocols designed for ad-hoc networks, namely MLE_-TPSN (i.e., TPSN [Ganeriwal et al., 2003] combined with MLE [Leng et al., 2010]), FTSP [Maróti et al., 2004], PulseSync [Lenzen et al., 2009], WMTS [He et al., 2014a] (a variant of MTS [He et al., 2014a]) and ATS [Schenato et al., 2011]. These protocols were proposed for wireless sensor networks and need modifications to be used on our target platform[8]. This section lists these modifications. Note that the modifications operated do not alter the general high-level framework of the compared protocols.

Communication Medium One of the adaptation is to consider a local and wired communication medium instead of a wireless and shared one. The main differences this adaptation causes, from a data-link point of view, are twofold. First, it entails the absence of message loss due to interferences/collisions on the communication medium. Second, in order to broadcast a message to all neighbors, a node has to send an individual copy of that message to all of them.

Communication Delay Compensation As explained in the state-of-the-art section, the methods used by these protocols to compensate for communication delays are not all directly applicable to our target platform. We recall that three methods are applicable to our target system, namely RTT, FD and PRED (see Section 4.3.3). TPSN is based on RTT. MLE_TPSN uses round-trip messages and computes the maximum likelihood estimation of the current global time on the last 5 synchronization points. We use FTSP with PRED because the method proposed in FTSP, which is highly accurate, is not applicable to our target system and because PRED is, on average, the most precise method for our system. PulseSync employs the same method as FTSP, thus we use PulseSync with PRED. In ATS, the authors suggest using the most precise method and utilizing FD for the experimental evaluation. Since PRED is, on average, more precise than FD in our system, we use ATS with PRED. We also use PRED to compensate for communication delays in WMTS.

The ATS and the WMTS Protocols ATS and WMTS are respectively average- and maximum-value consensus-based decentralized protocols. WMTS and ATS compensates for clock skew using averaging techniques. In WMTS and in the original version of ATS, modules use the last two clock readings of a neighbor to estimate its relative clock skew. In our modified version of ATS, we use the oldest and the newest clock readings to estimate the relative clock skew. This modification leads to better performance in our system. The ATS protocol takes input parameters, e.g., the probability of updating the clock offset and the clock skew of the modules at each synchronization round. We adopted the parameters used in the evaluation subsection of the original article [Schenato et al., 2011].

[8] Our implementation of these protocols is available online at: https://github.com/nazandre/thesis

The MRTP and the TPSN Protocols MRTP and TPSN are centralized protocols in which modules get periodically synchronized with the time master. In MRTP and TPSN, the time master is elected using an external algorithm and child modules are recursively synchronized by their parents along the edges of a spanning tree. For the leader election problem, we consider two algorithms. To elect the minimum-identifier module, we use the MIN_ID algorithm that we defined as MIN-ID in Section 3.9.3.1. To elect a central node, we use PC2LE-CENTER, the center version of the PC2LE framework introduced in Section 3.8. In the rest of the evaluation section, we use PC2LE to refer to PC2LE-CENTER for conciseness reasons. Table 4.7 shows the performance of these two election algorithms on our target systems. We use the CHEUNG-BFS-ST-CB algorithm presented in Section 3.5 to build the synchronization tree. In [Ganeriwal et al., 2003], the author states that any method can be used to select the time master and suggests that the minimum-identifier election algorithm presented in [Malpani et al., 2000] can be used. Thus, we use TPSN with MIN-ID. In addition, in the original version of TPSN, child modules overhear the messages exchanged during the synchronization process of their parent. As our platform uses contact communications, messages sent to a node cannot be overheard by other nodes. Thus, in our version of TPSN, we added an extra message sent by the parent to trigger the synchronization of child modules. Moreover, the modules use a linear model and MLE [Leng et al., 2010] to estimate the clock parameters. During a synchronization phase, modules only use the last timing information to disseminate the global time through the system. Without this last modification, MLE_TPSN diverges slowly in our simulations.

System	Algorithms	Simulated execution time (s)	Average number of messages (per module)	Elected-node eccentricity
Ball(5)	MIN_ID	0.25	38	10
	PC2LE	0.72	133	5
Ball(10)	MIN_ID	0.53	84	30
	PC2LE	1.96	420	17
Ball(27)	MIN_ID	0.83	107	54
	PC2LE	3.41	735	30

Table 4.7: Performance of election algorithms on the systems used for the evaluation of time synchronization protocols.

The FTSP and the PulseSync Protcols FTSP and PulseSync are centralized protocols in which modules get periodically synchronized with the time master. FTSP and PulseSync are infrastructure-less. During the synchronization phases, the minimum-identifier module gets implicitly elected as the time master. If a module has not received new synchronization messages for some synchronization periods (5 in our implementation), it declares itself time master and starts synchronizing the other modules. A module updates its belief concerning the current time master in the system whenever it receives a synchronization message advertising for a time master with a lower identifier.

In FTSP, a new time master ignores synchronization messages advertising for lower-identifier nodes during 3 synchronization periods. The FTSP protocol also takes as a parameter the number of synchronization messages that a node needs to have received before it considers itself to be synchronized and starts to synchronize neighboring nodes. In our simulations, we use the value of 3. For a better performance, we proceed to the subsequent modifications of FTSP, also suggested in [Lenzen et al., 2009]. In the original version of FTSP, synchronized modules ignore the received global time values that are too far from their own estimation of the current global time. As shown in the next subsection, FTSP does not provide precise synchronization in our target system and we had to suppress this filtering procedure in order to obtain better results. Additionally, in our version of FTSP, modules clear their linear regression table whenever they get synchronized by a new time master.

PulseSync accurately synchronizes nodes using rapid network-wide flooding. Sophisticated methods have been proposed to achieve fast flooding in WSN where messages may interfere and collide with each other (e.g., [Ferrari et al., 2011]). Our target system does not assume any specific mechanism to quickly disseminate a message through the network. Moreover, Blinky Blocks networks are not prone to message collisions. In our implementation of PulseSync, synchronization messages are handled like any other message. In particular, messages are not prioritized in message queues.

Naming Convention We use the following format to name the different approaches compared: [ORIGINAL PROTOCOL NAME]-[LEADER ELECTION ALGORITHM]-[COMMUNICATION DELAY COMPENSATION METHOD]. For instance, MRTP-PC2LE-PRED refers to the MRTP synchronization protocol based on the PC2LE leader election algorithm and our predictive model to compensate for communication delays.

4.7.2.2/ Time of Convergence and Achievable Precision

Figure 4.17 shows the average maximum pairwise synchronization error of the modules over time for the compared synchronization protocols. During the first hour, the modules were not synchronized and progressively drifted apart. The system reached a synchronization error of more than 40 seconds.

MRTP, MLE_TPSN and PulseSync centralized protocols converge in a few seconds in the three systems. We recall that MRTP and MLE_TPSN first elect a leader, build a spanning tree, and then start synchronizing the modules. In PulseSync, modules wait for 5 synchronization periods (i.e., 25 seconds) without hearing a synchronization message before declaring themselves time masters and trying to synchronize the other nodes. This mechanism causes PulseSync to converge slightly more slowly but makes this protocol inherently tolerant of faults.

As expected, ATS, which is an average consensus-based decentralized protocol, converges much more slowly and the time of convergence significantly increases with the system size. In Ball(15), ATS converges only after about 30 minutes of periodic syn-

chronization. WMTS, which is a maximum-value consensus-based protocol, converges more quickly than ATS. But WMTS is still slightly slower than the MRTP, MLE_TPSN and PulseSync centralized protocols.

FTSP does not converge in large ensembles of Blinky Blocks. Theoretically, FTSP should have converged in less than 15 minutes in Ball(27) [Maróti et al., 2004]. As explained in the related work subsection, the FTSP synchronization waves are slowly flooded through the network using asynchronous broadcasts, whereas in MRTP, MLE_TPSN and PulseSync, the current global time gets quickly disseminated throughout the entire network. This last scheme significantly reduces the impact of clock inaccuracies (due to noise, skew variations, time-increasing errors in the local estimation of the global time) on the synchronization precision and the time of convergence.

Figure 4.18 shows statistics on the maximum pairwise synchronization error after convergence. Unsurprisingly, the synchronization precision of all the protocols decreases with the network size. MRTP, MLE_TPSN and PulseSync, which are centralized protocols, have a synchronization precision of a few dozen milliseconds in all the systems considered.

MRTP-PC2LE-PRED is the most precise protocol. As shown in Figure 4.18, using a central node as the time master improves the average maximum pairwise synchronization error of MRTP by about 0.6 to 3.5 milliseconds in the different ball systems (MRTP-PC2LE-PRED vs MRTP-MIN_ID-PRED). Moreover, the precision improvement increases with the diameter of the ball.

Unsurprisingly, MRTP-MIN_ID-PRED and PulseSync-PRED have, on average, a similar synchronization precision. It was awaited as the two protocols only differ by the mechanism they use to elect the minimum-identifier node and by their infrastructure (i.e., MRTP uses a breadth-first spanning tree while PulseSync is infrastructure-less and floods the network). However, it must be noted that MRTP-MIN_ID-PRED has a slightly lower worst-case synchronization error. We did not investigate this point but we suspect this could be due to the fact that, in MRTP, a node always gets synchronized by a message that has traveled on the same and shortest path while in our implementation of PulseSync synchronization messages can come from different and possibly not shortest paths, depending on the network traffic.

MRTP-MIN_ID-PRED is on average about 2.3 milliseconds more precise than MLE_TPSN in Ball(27) but has a 2-millisecond higher worst-case synchronization error in both the Ball(15) and Ball(27) systems. We recall that these two protocols only differ by the method they use to compensate for communication delays and clock skew. Moreover, as shown in the next subsection, MRTP with PRED is more communication-efficient than MLE_TPSN.

As announced in subsection 4.7.1.2, MRTP can effectively synchronize the Ball(27) system, composed of 27,775 modules, to less than 40 milliseconds. Indeed, it synchronizes this system to 17 milliseconds on average and to 24 milliseconds at worst.

As expected, the ATS decentralized method is less precise than the centralized ones. The WMTS decentralized method exhibits similar synchronization precision in Ball(5) to that of the centralized methods, while it fails to accurately synchronize the Ball(27) system.

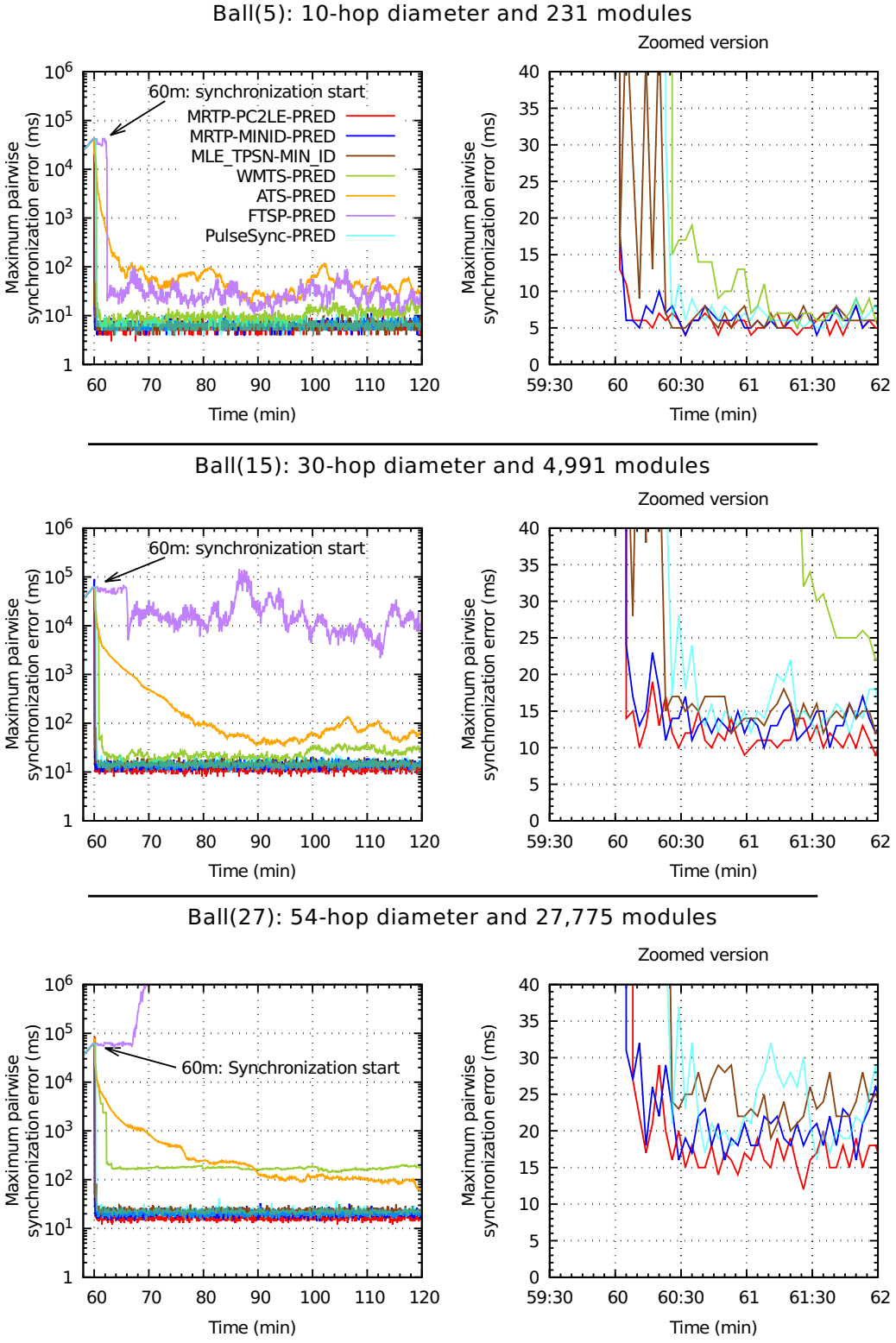


Figure 4.17: Maximum pairwise synchronization error over time. This figure shows both the time of convergence and the achievable precision for each protocol on the different Ball systems.

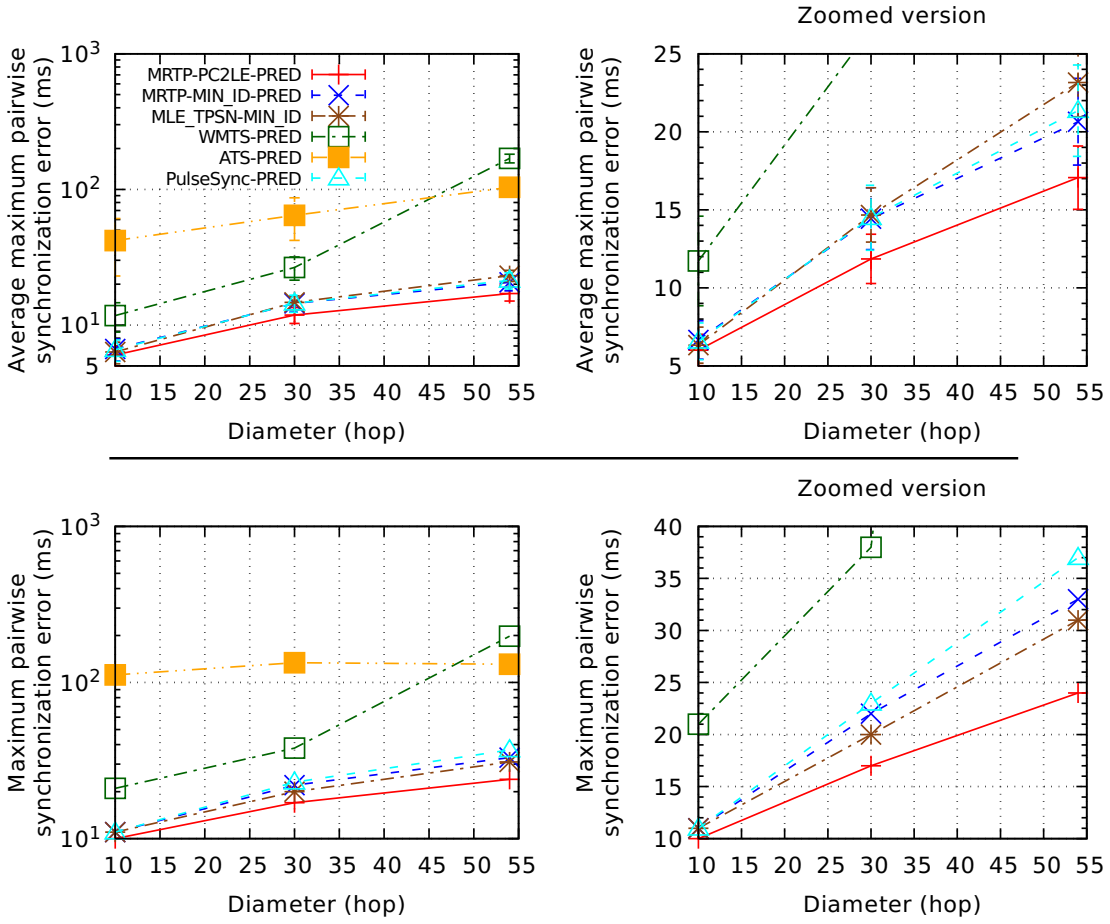


Figure 4.18: Synchronization precision. At the top, average maximum pairwise synchronization error in the last 30 minutes of the experiment (± standard deviation). At the bottom, the maximum pairwise synchronization error.

4.7.2.3/ Communication Efficiency

Figure 4.19 shows the average number of messages sent per module and its decomposition according to the message types. We consider three types of message: the messages due to the leader election process, the ones due to the tree infrastructure creation and the synchronization messages.

As expected, ATS and WMTS decentralized synchronization protocols use on average more messages per module than the MRTP, MLE_TPSN and PulseSync centralized protocols. In addition, PulseSync, which uses network-wide floodings, generates on average more messages per module than MRTP and MLE_TPSN that use a tree-like structure. Thus, the message cost induced by both the leader election process and the infrastructure construction is compensated for in less than one hour.

Let k denote the number of messages used by the compensation delay method ($k = 1$ for PRED and $k = 3$ for round-trip-time-based methods as in MLE_TPSN). In decentralized

methods, $2km$ messages[9] are sent per synchronization round, while $k(n-1)$ messages[10] are sent in MRTP and MLE_TPSN, and $2m-(n-1)$ messages[11] are sent in PulseSync-PRED (after the implicit time-master election has converged). We recall that $n-1 \leq m$ in connected networks. As MLE_TPSN uses a round-trip time based method, it generates three times more synchronization messages per synchronization phase than MRTP with PRED. In compact systems, the number of links is more important than the number of nodes. Thus, in these systems, PulseSync generates more messages per synchronization round than MRTP with PRED. However, PulseSync is inherently more tolerant of network failures because synchronization waves are flooded through all links and not only along the links of a spanning tree. Thus, if a link fails but the system remains connected, PulseSync may still be able to synchronize all the modules. Nevertheless, in a spanning tree, if a link fails, all the nodes of a sub-tree will not get synchronized.

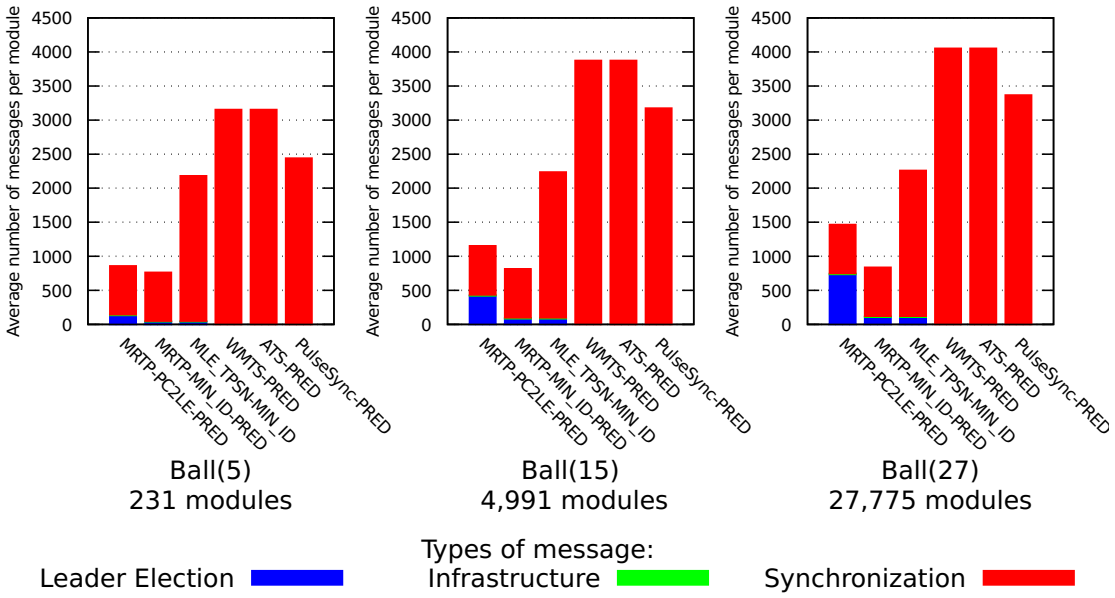


Figure 4.19: Average number of messages sent per module in time synchronization protocols.

We measured the maximum message queue size reached by the modules taking into account both the incoming and the outgoing messages. We observed that for any module, the ratio of the maximum queue size reached to the number of neighbors of that module, remains below or equal to three, regardless of the size of the networks for all the protocols, except for PulseSync. For PulseSync, the ratio reached the value of 4.5. This is due to the uncontrolled-broadcast problem explained in Section 3.5.1.2. This issue does not have a big impact in this case. Indeed, because clocks are drifting apart, nodes trigger synchronization waves at slightly different instants. Thus, neither the leader election

[9] Every module sends a synchronization message to all its neighbors.
[10] Every module except the time master gets synchronized by a single node.
[11] Every module sends a synchronization message to all its neighbors except to the one from which it got synchronized.

process, which involves network-wide flooding(s), nor the actual synchronization phases overwhelm the network. The traffic generated by the synchronization protocols remains well controlled and modules do not require a lot of memory space to store incoming and outgoing messages.

4.8/ DISCUSSION

MRTP is intended to synchronize large-scale and fairly stable systems where changes in the network topology, due for instance to module mobility, or potential module or link failures, are infrequent. Our protocol achieves its performance by combining several mechanisms: distributed central-time-master election, fast and recursive propagation of synchronization waves along the edges of a breadth-first spanning tree, low-level timestamping and per-hop compensation for communication delays using the most-appropriate method for the target platform, and clock skew compensation using linear regression.

Design Choices In MRTP, a dynamically elected central module periodically synchronizes the system. We assume the network traffic to be evenly distributed in the network. Placing the time master close to the center of the network increases the overall synchronization precision because cumulative errors are made every hop. This strategy is particularly judicious in our context because large-scale modular robots with neighbor-to-neighbor communications tend to exhibit long hop distances. In order to synchronize the system, the time master periodically launches synchronization waves, which are recursively propagated along the edge of a breadth-first spanning tree. Slave modules propagate these waves to their children in the tree shortly after reception. As explained in [Lenzen et al., 2009], optimal synchronization requires a fast propagation scheme. Also note that using a tree is more communication-efficient in compact systems than flooding approaches. Indeed, since there is no broadcast support in the neighbor-to-neighbor communication model, a node has to send an individual copy of a message to all its neighbors in order to broadcast that message. Furthermore, using a breadth-first tree guarantees that synchronization messages always travel on the same and shortest paths. This also leads to better synchronization precision. MRTP performs per-hop synchronization, i.e., a module gets synchronized by a one-hop neighbor. At each hop, the propagated estimation of the current global time is updated to take into account communication delays and time of residence in intermediate modules. Any approach to compensate for these delays can be used in MRTP. Most of the existing approaches use low-level timestamping to suppress the main sources of uncertainty in delay estimations. The best-suited technique to be actually used in MRTP depends on the target platform (i.e., the clock precision, its resolution, the communication mechanism and the network load) and should be carefully selected, since it has a direct impact on the performance of our protocol, both in terms of precision and communication efficiency. We provided a method to experimentally evaluate the precision of a given approach over multiple hops.

Network Density We showed that, with a central time master, MRTP can synchronize the 54-hop-diameter ball system composed of 27,775 modules to 24 milliseconds, at worst. In Section 3.3.2, we showed that MRTP can synchronize a sparser 83-hop-diameter system, composed of 1,456 nodes, to 29 milliseconds, at worst, when the time master is placed at the center of the system. These worst-case synchronization errors (i.e., the maximum value over all the maximum pairwise synchronization error values that were captured every 3 seconds) are consistent with each other. However, we observed that the averaged maximum pairwise synchronization error is smaller on the sparser system (13 milliseconds versus 17 milliseconds), although it exhibits a larger diameter. We did not investigate this phenomenon, but we believe this is because bad cases happen more rarely in the sparser system, in part because there are less nodes and less long independent paths in that system. Indeed, nodes that receive synchronization messages that have traveled on long and almost independent paths, causing the error accumulated at every hop to be propagated differently, tend to exhibit a high maximum pairwise synchronization error.

Moreover, it must be noted that, on the Blinky Blocks, the number of simultaneous communications impacts the transfer time (see Section 4.6.2). Thus, methods of compensating for communication delays may exhibit a different precision depending on the network density, as shown in Section 4.7.1.3. Hence, the achievable synchronization precision may differ depending on the network density.

Portability Our protocol is portable to any modular robot system where modules interact together using only neighbor-to-neighbor communications even if their internal clocks are low precision and have high skew relative to one another. Depending on the time master election procedure, it may also be required that every module has a unique identifier. We evaluated MRTP on the Blinky Blocks platform, which is equipped with a very low-accuracy and poor-resolution clock, but it must be noted that our protocol can also be used in systems with more precise clocks. It will indeed have two main effects. First, a lower resolution will lead to more precise local clock readings, i.e., more precise message timestamps. Hence, communication delays may be more precisely captured and compensated for, using potentially a different method than the predictive one we use with the Blinky Blocks. Second, a more precise clock implies reduced clock skew, drift (variation of skew) and noise. This can only increase our protocol precision. It must be noted that, even with higher-precision clocks, it is still appropriate to use a linear model to compensate for short-term clock skew. Indeed, this approach is also commonly used in systems equipped with more precise clocks (e.g., RBS [Elson et al., 2002], FTSP [Maróti et al., 2004], PulseSync [Lenzen et al., 2015], etc.). Consequently, our protocol should also be able to efficiently synchronize systems equipped with higher-precision clocks. We let the evaluation of our protocol in such systems for future works.

4.9/ CONCLUSION

In this chapter, we described the Modular Robot Time Protocol (MRTP), a network-wide time synchronization protocol for modular robots. We evaluated our protocol on the Blinky Blocks platform, both on hardware and through simulations. We showed that MRTP can potentially manage systems composed of up to 27,775 Blinky Blocks. Furthermore, the experimental results show that MRTP is able to successfully maintain a Blinky Blocks system synchronized to a few milliseconds, using few network resources at runtime, although the Blinky Blocks are equipped with very low-accuracy and poor-resolution clocks. Simulations results show that MRTP exhibits on average a lower maximum pairwise synchronization error than the compared protocols, while sending more than half less messages in compact systems.

5

Modular Robot Self-Reconfiguration

Contents

5.1/ INTRODUCTION

The most studied algorithm in Modular Self-Reconfigurable Robots (MSRs) is the self-reconfiguration algorithm which causes the modules to move from one configuration (the *initial shape*) to another (the *goal shape*) (see Figure 5.1).

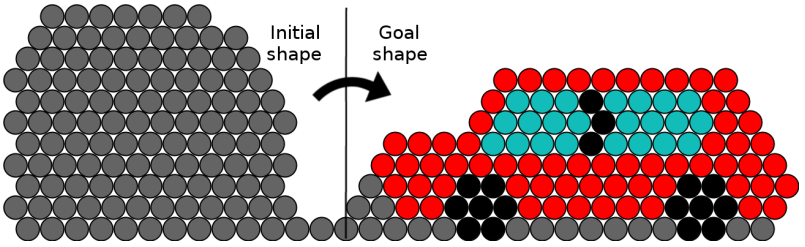


Figure 5.1: Example of initial and goal shapes. Self-reconfiguration is the process during which the initial clump of modules on the left self-reconfigures into the car shape on the right.

As explained in the Introduction chapter, self-reconfiguration algorithms pose several algorithmic challenges. In the first place, planning is challenging as the number of possible unique configurations is huge and the exploration space between two random configurations is exponential in the number of modules, due to potential concurrent moves. This prevents us from finding a complete optimal planning for all but the simplest configurations. In addition to the path-planning problem, the distributed coordination of mobile autonomous modules connected in time-varying ways is also a challenging issue. In particular, modules have to coordinate their motions in order not to collide with each other.

Self-reconfiguration algorithms are tailored for a specific class of modular robots, with specific motion constraints [Stoy et al., 2011], for example using cubes sliding on the floor, some motions need a cooperation process that complicates motion algorithms [Piranda et al., 2016a]. In this work, we consider the 2D Catoms (see Chapter 2). Our assumptions and system model are detailed in Section 5.2).

The contribution of this chapter is to propose the Cylindrical-Catoms Self-Reconfiguration (C2SR) algorithm[1] which is asynchronous, deterministic, fully decentralized and able to manage almost any kind of initial and goal compact shapes (see Section 5.4). Although our work is focused on the algorithm, we carry out our analysis with respect to the hardware constraints of the 2D Catoms. C2SR is inspired by the algorithm in [Rubenstein et al., 2014] proposed for swarm robotic systems which assume different physical constraints. C2SR is a step toward realizing programmable matter.

We implemented our algorithm and evaluated it through simulations with VisibleSim. We show the effectiveness of C2SR on large-scale ensembles composed of up to ten thousand modules. We also show the effectiveness of our algorithm and study its performance in terms of communications, movements, and execution time.

The rest of this chapter is organized as follows. In section 5.2, we define the system model

[1] Some simulations of self-reconfiguration with C2SR are available online in video at https://youtu.be/XGnY-oS4Nw0

and assumptions. Afterwards, we discuss the related work in section 5.3. In section 5.4, we present the general idea of C2SR and in section 5.5, we describe its implementation. In section 5.6, experimental results are presented and analyzed. Section 5.7 concludes this chapter.

5.2/ System Model and Assumptions

In this chapter, we consider the 2D Catoms. This section presents our assumptions and system model.

We assume that 2D Catoms are organized in a horizontal pointy-topped hexagonal lattice where modules have up to six neighbors. Modules can communicate together using neighbor-to-neighbor communications. We assume that modules automatically discover their neighbors, using communications after becoming attached. We consider that moving modules cannot communicate with any other module. $\mathcal{N}_{C_i}^N$ denotes the network neighbors of the module C_i. Catoms on the periphery have Clockwise (CW) and Counter-Clockwise (CCW) neighboring Catoms that also belong to the periphery. For instance, in Figure 5.2, C_9 is the Clockwise (CW) peripheral neighbor of C_6 and the Counter-Clockwise (CCW) neighbor of C_{10}. C_{11} is both the CW and CCW peripheral neighbor of C_{12}.

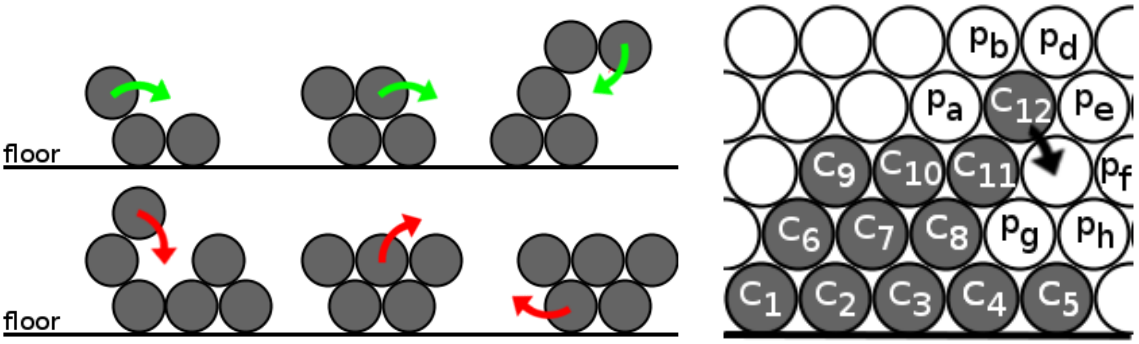


Figure 5.2: On the left, motion constraints in our model: examples of feasible (at the top) and infeasible moves (at the bottom). On the right, a labeled system: gray cells are occupied by a module, whereas white cells are empty. Some of the empty cells are labeled with their position (e.g., p_a, p_b, etc.).

$p_{C_i} = (x_{C_i}, y_{C_i})$ denotes the coordinates of the 2D Catom C_i in the horizontal hexagonal lattice. $p_{C_i}.x$ denotes the column of C_i in the lattice, while $p_{C_i}.y$ denotes the row of C_i. For instance, in Figure 5.2, $p_{C_2}.y = 0$ and $p_{C_9}.y = 2$. We assume that, at any time, modules know both their coordinates in the lattice and the coordinates of their neighbor through an external algorithm, e.g., [Funiak et al., 2009] or a distributed and incremental version of [Moffo et al., 2016].

Moreover, a 2D Catom can roll CW or CCW around a stationary module. During an atomic move, a module rotates 60, going from one cell of the lattice to its adjacent cell. We assume that a 2D Catom has only the capability to lift itself, it cannot carry or push

other modules. A module can move if it satisfies the freedom of movement rule (see Rule 1).

Rule 1: the freedom of movement rule. Because of possible mismatching issues due to physical constraints, a 2D Catom can only move from/into a cell if this cell is currently unoccupied and if no two symmetrically opposing cells adjacent to that cell are occupied (see Figure 5.2). Furthermore, we consider the floor as if it were filled with 2D Catoms. If a 2D Catom, C_i, satisfies the freedom of movement rule, $free(C_i)$ is true, otherwise it is false.

We assume that 2D Catoms are not provided with any hardware mechanism to handle collision. Thus, collisions have to be prevented by the self-reconfiguration algorithm, using communications.

We use $\mathcal{N}_p^K$ to denote the set of modules geographically adjacent to position p. A module C_i, moving from p_{C_i} to p'_{C_i}, is somewhere between these two positions, and thus, C_i belongs to the set of geographically adjacent modules of all the cells adjacent to p_{C_i} or p'_{C_i}. For instance, in the labeled system depicted in Figure 5.2, module C_{12} is moving and thus it belongs to $\mathcal{N}_{p_a}^K$, $\mathcal{N}_{p_b}^K$, $\mathcal{N}_{p_d}^K$, $\mathcal{N}_{p_{C_{12}}}^K$, $\mathcal{N}_{p_e}^K$, $\mathcal{N}_{p_{C_{11}}}^K$, $\mathcal{N}_{p'_{C_{12}}}^K$, $\mathcal{N}_{p_f}^K$, $\mathcal{N}_{p_g}^K$ and $\mathcal{N}_{p_h}^K$. Note that in the presence of moving modules, $\mathcal{N}_{p_{C_i}}^K$ may be different from $\mathcal{N}_{C_i}^N$. Also notice that the construction of the $\mathcal{N}^K$ sets is not automatic. 2D Catoms are not equipped with any presence sensor. Maintaining on Catoms the $\mathcal{N}^K$ set of some specific nearby positions, using only communications, is one of the key operations in the implementation of our distributed algorithm.

$\mathcal{I}$ and $\mathcal{G}$ denote the initial and the goal shapes, respectively. Our algorithm assumes some admissibility conditions for $\mathcal{I}$ and $\mathcal{G}$ (see Section 5.4). We also consider that every module stores a representation of the shape geometry of $\mathcal{G}$. The goal shape can be stored efficiently using Constructive Solid Geometry for Programmable Matter (CSG4PM) [Tucci et al., 2017]. CSG4PM encodes a vectorial representation of a shape using operations on primitive shapes. The shape can be scaled up without increasing the memory usage to store it. Moreover, CSG4PM provides processing-efficient methods of checking whether a given lattice cell belongs to the described shape or not. For instance, CSG4PM requires only 233 bytes to store goal configurations (shape and colors) similar to the car configurations in Figures 5.1 and 5.8, which are respectively composed of 120 and 9,644 modules.

Note that colors are used for illustration purposes only. The current prototype is not equipped with any mechanism to glow with color. It is possible to do so, but the weight of that color mechanism will probably change the 2D Catom motion speed (see Section 2.3.2).

Furthermore, we assume a failure-free environment, i.e., we assume there is no module, communication, move or lattice failure during the algorithm execution.

5.3/ STATE OF THE ART

Self-reconfiguration and self-assembly have attracted a lot of attention in the last two decades. Algorithms have been proposed for modules of different shapes, with different physical motion constraints and arranged in various ways. In this chapter, we only consider the self-reconfiguration of systems composed of elements organized in a vertical and two-dimensional hexagonal lattice. Algorithms also differ by their restrictions on the initial and goal shapes. Our algorithm can manage almost any kind of initial and goal compact shapes (see Section 5.4). Algorithms also vary in their control properties. In particular, they can be centralized or distributed, and synchronous or asynchronous.

In [Walter et al., 2000], the authors propose a distributed algorithm to perform chain-to-chain self-reconfiguration in a hexagonal lattice. Modules move in synchronous rounds. This work was later extended in order to allow self-reconfiguration from a chain configuration into an arbitrary shape with some admissibility conditions [Walter et al., 2005, Bateau et al., 2012]. These algorithms assume less restrictive motion constraints than those we assume for the 2D Catoms. For instance, these algorithms allow the first two motions described as infeasible in Figure 5.2, starting from the left.

In [Hurtado et al., 2013], Hurtado et al. propose a self-reconfiguration algorithm for modular robots arranged in a two-dimensional square or a two-dimensional hexagonal lattice. This algorithm is intended to run in a synchronized framework. The proposed method runs in two stages. It first reconfigures the robot from the initial configuration into a strip configuration and then from the strip configuration into the goal shape. A leader assigns a final destination location for every module in the strip configuration using a tree-based approach.

Self-reconfiguration presented in [Lakhlef et al., 2014, Lakhlef et al., 2015b, Lakhlef et al., 2015a] consists in using map-less representation for describing shapes. The benefit lies in a reduced memory footprint, but the number of supported goal shapes is limited. Proposed distributed algorithms manage to construct square shapes with spherical modules arranged in a two-dimensional hexagonal lattice. In [Lakhlef et al., 2015b, Lakhlef et al., 2015a], the authors propose to self-reconfigure a chain of modules into a square shape and demonstrate that the number of movements can be predicted. A more general algorithm designed to self-reconfigure arbitrary connected shapes into a square shape is presented in [Lakhlef et al., 2014].

Algorithms allowing the reconfiguration of an initial clump of modules arranged in a hexagonal lattice into a chain configuration were proposed in [Wong et al., 2013, Wong et al., 2015]. These algorithms do not require message passing and do not use any preprocessing. In these algorithms, modules can both rotate and slide over other modules. Thus, these algorithms assume less restrictive motion constraints than ours.

In [De Rosa et al., 2006], the authors propose a distributed shape formation algorithm based on hole motions, for ensembles arranged in a hexagonal lattice. This algorithm can construct various shapes by randomly moving empty spaces within the ensemble. Although a wide variety of shapes can be built, this algorithm requires less restrictive motion constraints than ours, e.g., it allows the first two infeasible motions in Figure 5.2.

In [Rubenstein et al., 2014], the authors propose a parallel, decentralized and asynchronous algorithm for the Kilobot swarm system [Rubenstein et al., 2014] to self-assemble almost any kind of compact two-dimensional shapes. This algorithm has been applied to hardware systems with more than a thousand individual robots per swarm entity. However, these swarm robots have different physical motion constraints. During the self-assembly process, Kilobots may collide with one another. While this is possible with Kilobots, this is not acceptable in our system.

Table 5.1 summarizes the related work. Existing algorithms contain interesting ideas but consider different physical motion constraints, different restrictions on the initial and goal shapes and different control properties. The contribution of this chapter is to propose a distributed, fully decentralized, asynchronous and parallel self-reconfiguration algorithm for 2D Catoms that can manage almost any kind of initial and final compact shapes.

Cite	Shapes	Module movement capabilities	Collision and deadlock avoidance
[Walter et al., 2000]	chain to chain	relaxed	centralized pre-computation, synchronous rounds
[Walter et al., 2005, Bateau et al., 2012]	chain to 2D	relaxed	centralized pre-computation, synchronous rounds
[Hurtado et al., 2013]	2D	UN	synchronized framework, single direction, intermediate configuration and priority numbers
[Lakhlef et al., 2015b, Lakhlef et al., 2015a]	chain to square	relaxed and very relaxed	predefined shape construction
[Lakhlef et al., 2014]	arbitrary connected to square	relaxed	predefined shape construction
[Wong et al., 2015]	compact 2D to chain	relaxed	touch sensors, synchronous rounds, single direction
[De Rosa et al., 2006]	2D	relaxed	UN
[Rubenstein et al., 2014]	horizontal 2D compact	very relaxed	collision allowed (swarm robotic)
Our Contribution: C2SR	vertical 2D compact	strict	messages, single direction

Table 5.1: Summary of the state of the art on self-reconfiguration in MSRs where modules are arranged in a hexagonal lattice. “UN” stands for “Unknown”.

5.4/ C2SR ALGORITHM AT A GLANCE

In this section, we present the general idea of the Cylindrical-Catoms Self-Reconfiguration (C2SR) algorithm that reconfigures a robot composed of modules from an initial shape $\mathcal{I}$ into a goal one $\mathcal{G}$.

Both shapes have to satisfy some admissibility conditions. We provide some intuitions about them in this paragraph and in Figure 5.3. A more formal description of the conditions and their demonstration are left for future work. Both shapes are compact, i.e., they do not contain holes, they are homeomorphic to a sphere. Moreover, both shapes are next to each other and intersect in one or more bottom cells. Let the peripheral path be the path formed from the empty cells on the periphery of both shapes, starting from and ending at the second horizontal layer (see Figure 5.3). This path has to be large enough to allow some modules, which progress along that path in the same direction with an empty space of at least one cell between successive modules, to move without violating our motion constraints and without risking colliding/getting attached to one another (see Figure 5.3 and Rule 1). Note that this condition implies that, in the upper layers, the horizontal space between the initial shape and the goal shape has to be sufficiently large to enable these modules to move between the two shapes. Furthermore, the number of 2D Catoms in $\mathcal{I}$ has to be greater than or at least equal to the number of target positions in $\mathcal{G}$ (i.e., $|\mathcal{I}| \geq |\mathcal{G}|$).

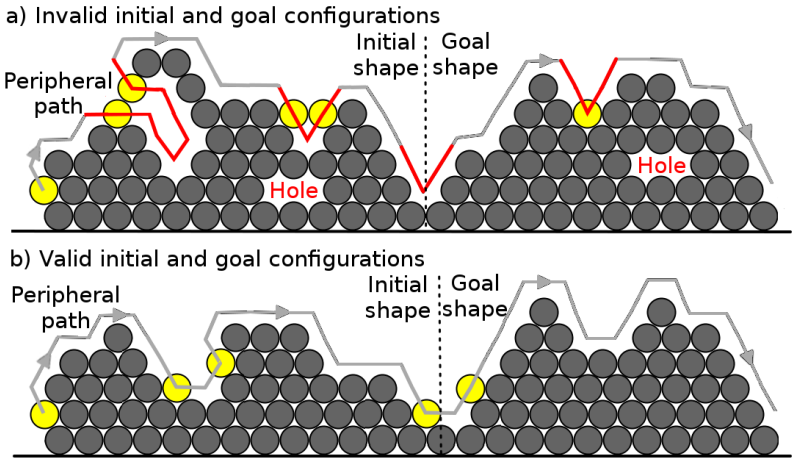


Figure 5.3: Invalid (at the top) and valid (at the bottom) initial and goal configurations in C2SR. Modules in yellow, which are not part of the initial or goal shapes, progress along the peripheral path in the same direction with an empty space of at least one cell between successive modules. The configurations at the top are not valid for several reasons. First, they do not intersect in at least one cell. Second, they both contain a hole. Third, the peripheral path is not large enough at the locations in red. Indeed, the modules in yellow could not move without violating our motion constraints and without getting attached to each other.

During the execution of C2SR with shapes individually composed of only continuous horizontal layers, the goal shape is progressively constructed from the bottom layer to the

top one by stripping the initial shape, module by module in reverse order (see Figure 5.4). Because of physical constraints, at a given instant, only modules on the periphery can move. In order to avoid module collisions and deadlocks, peripheral modules form a stream: modules roll in the same direction d (CW in Figures 5.1 and 5.4), and maintain an empty cell between one another using message exchanges. Modules in the stream do not overtake one another.

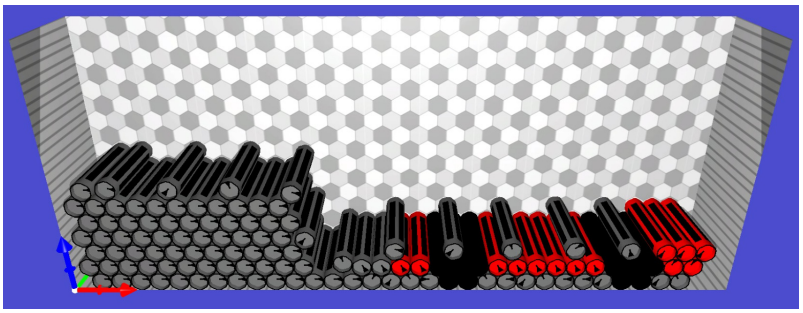


Figure 5.4: Screenshot during the self-reconfiguration process using C2SR with the initial and goal shapes of Figure 5.1. The modules in the stream progress by rotating CW.

C2SR is based on a set of states and transition rules. Figure 5.5 shows the state diagram of our algorithm. Modules can have different states, namely INIT, BLOCKED, WAITING, MOVING, or GOAL. Modules are initially in the INIT state. Modules in the WAITING and MOVING states belong to the stream. Figure 5.6 shows the different states of the modules in Figure 5.4.

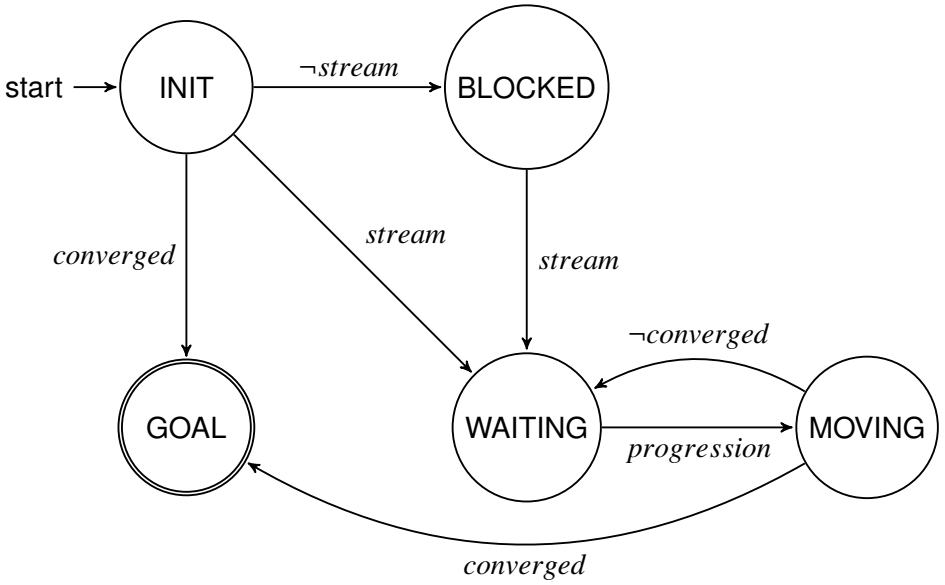


Figure 5.5: C2SR state diagram.

A module locally decides to start taking part in the stream if it satisfies the stream entrance rule (see Rule 2). Intuitively, a free module enters the stream if moving in the direction d consists in: moving around a module on the ground or descending $\mathcal{I}$.

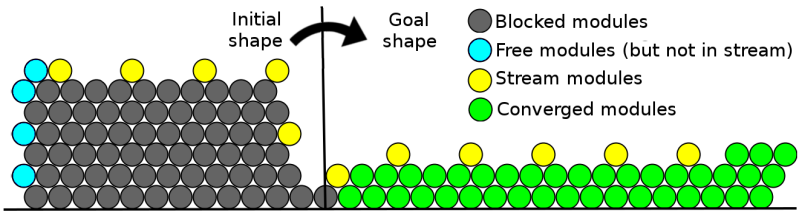


Figure 5.6: Different module states in C2SR. Note that, at this particular moment of the reconfiguration, no Catom is in the moving state.

Rule 2: the stream entrance rule. Let us consider two modules C_i and C_j such that both C_i and C_j are on the periphery and C_j is the next peripheral neighbor of C_i in the direction of rotation, d. p'_{C_i} denotes the position that C_i would occupy after its rotation around C_j. C_i decides to take part in the stream if the following logical condition is satisfied:

$$
\begin{aligned}
stream(C_i) :- \; & state(C_i) \neq \text{GOAL} \text{ // has not converged yet} \\
\wedge \; & free(C_i) \text{ // mechanical constraints} \\
\wedge \; & ((p_{C_i} \notin \mathcal{G} \wedge p_{C_j}.y = 0) \text{ // move around a module on the ground} \\
\vee \; & (p_{C_i} \notin \mathcal{G} \wedge p'_{C_i}.y \leq p_{C_i}.y)) \text{ // descend } \mathcal{I}
\end{aligned}
$$

A module in the stream decides to move if it satisfies the stream progression rule (see Rule 3). More precisely, a module in the stream can move if the set of modules geographically adjacent to its destination cell contains no more than three modules and none of them, except the module itself, belongs to the stream (see Figure 5.7). This rule requires local interactions with neighbors adjacent to its source and destination positions. These modules are at most two cells away. The admissibility conditions on $\mathcal{I}$, combined with the two rules above, guarantee that these modules are five network hops away at most.

Rule 3: the stream progression rule. A module C_i can move from its position p_{C_i} to the position p'_{C_i} if the following condition is satisfied:

$$
\begin{aligned}
progression(C_i) :- \; & state(C_i) = \text{WAITING} \text{ // in the stream} \\
\wedge \; & |\mathcal{N}^K_{p'_{C_i}}| \leq 3 \text{ // no more than 3 modules near the destination cell} \\
& \text{// no other stream module in the surroundings of the destination cell:} \\
\wedge \; & \nexists C_j \in \mathcal{N}^K_{p'_{C_i}} \mid C_j \neq C_i \wedge (state(C_i) = \text{WAITING} \vee state(C_i) = \text{MOVING})
\end{aligned}
$$

Rule 3 prevents collisions. The admissibility conditions on $\mathcal{I}$ and $\mathcal{G}$, combined with Rules 2 and 3, prevent deadlock. Note that, because of the stripping order and the construction order, our algorithm also guarantees that, at all times the system remains connected.

Each module checks for convergence using Rule 4 at initialization and after every move. A module has converged if it is initially in a goal position, or if it has reached $\mathcal{G}$ and moving in the direction d will cause it to leave $\mathcal{G}$ or to go up.

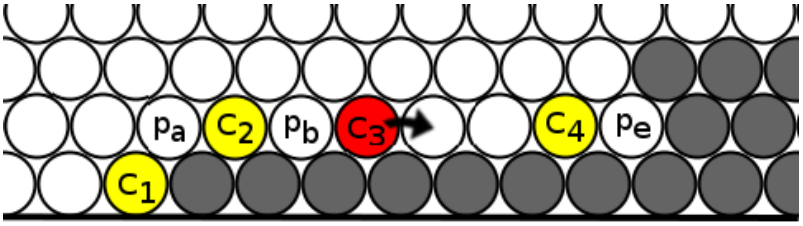


Figure 5.7: C2SR stream progression rule: a simple example. Modules should rotate CW. White cells are empty and some of them are labeled with their position in the lattice (e.g., p_a, p_b, etc.). Modules C_1, C_2, C_3 and C_4 are in the stream. C_3 is moving. C_1 cannot move because C_2 is in the stream and $C_2 \in \mathcal{N}^K_{p_a}$. C_2 cannot move because C_3 is in the stream and $C_3 \in \mathcal{N}^K_{p_b}$. C_3 can move to p'_{C_3} because $\mathcal{N}^K_{p'_{C_3}}$ contains only three modules and none of them is in the stream, except for C_3. C_4 cannot move because $|\mathcal{N}^K_{p_e}| = 5$.

Rule 4: the local convergence rule. Let us consider two modules C_i and C_j such that both C_i and C_j are on the periphery and C_j is the next peripheral neighbor of C_i in the direction of rotation. p'_{C_i} denotes the position that C_i would occupy after its rotation around C_j. C_i has converged if it satisfies the following condition:

$$\begin{aligned} converged(C_i) :- & (state(C_i) = \text{INIT} \wedge p_{C_i} \in \mathcal{G}) \text{ // initially in } \mathcal{G} \\ & \vee (p_{C_i} \in \mathcal{G} \wedge p'_{C_i} \notin \mathcal{G}) \text{ // about to leave } \mathcal{G} \\ & \vee (p_{C_i} \in \mathcal{G} \wedge p'_{C_i} \in \mathcal{G} \wedge p'_{C_i}.y > p_{C_i}.y) \text{ // about to go up in } \mathcal{G} \end{aligned}$$

Applying these rules in a distributed asynchronous system with parallel communications and motions is challenging. It is especially complex to maintain $\mathcal{N}^K$ sets using only communications. A complete implementation that overcomes this challenge is presented in the next section.

5.5/ C2SR Implementation

In this section, we provide a detailed implementation of C2SR[2]. Algorithm 14 shows the input and local variables of C2SR along with its initialization pseudo-code. Every module knows its position in the lattice, the goal shape, $\mathcal{G}$, and the rotation direction, d. Algorithm 15 describes some helper functions used in the description of our implementation of C2SR. Algorithm 16 provides the message handler pseudo-code of C2SR. Algorithm 17 gives the pseudo-code executed by a module after it has finished an atomic move. We assume that interrupts are disabled during message and event handler execution.

At initialization and during the execution, modules locally decide their state using Rules 1, 2 and 4. Modules in the stream move in the rotation direction d around their peripheral neighbor in the d direction. Before moving, modules have to ensure that the

[2]The complete source code of C2SR is available online at: https://github.com/nazandre/thesis

Input :
p_{C_i} // position of C_i
$d \in \{CW, CCW\}$ // direction of rotation
$\mathcal{G}$ // goal shape
Local Variables :
state // state of C_i
Movings // cells from/into which a neighbor module is moving
Pendings // pending clearance requests
clearance // clearance for the current move (if any)

1 **Initialization** of C_i:
2 $Movings \leftarrow \emptyset$; $Pendings \leftarrow \emptyset$; $clearance \leftarrow \perp$;
3 **if** $p_{C_i} \in \mathcal{G}$ **then**
4 $\quad state \leftarrow$ GOAL;
5 **else if** $isInStream()$ **then**
6 $\quad state \leftarrow$ WAITING;
7 $\quad requestClearance()$;
8 **else**
9 $\quad state \leftarrow$ BLOCKED;

Algorithm 14: C2SR algorithm input, local variables and initialization detailed for any module C_i.

1 **Function** *hasConverged()*:
$\quad$ // The local convergence rule (Rule 4)
2 $\quad$ **return** converged(C_i);

3 **Function** *areAdjacentCells(p_1, p_2)*:
4 $\quad$ **return** true if cells at positions p_1 and p_2 are adjacent in the hexagonal lattice, false otherwise;

5 **Function** *oppositeDirection(d)*:
$\quad$ // $d \in \{CW, CCW\}$
6 $\quad$ **return** the opposite direction of d;

7 **Function** *isFree()*:
$\quad$ // The freedom of movement rule (Rule 1)
8 $\quad$ **return** free(C_i) considering both $\mathcal{N}_{C_i}^N$ and $Movings$;

9 **Function** *isInStream()*:
$\quad$ // The stream entrance rule (Rule 2)
10 $\quad$ **return** stream(C_i) considering both $\mathcal{N}_{C_i}^N$ and $Movings$;

11 **Function** *getNeighbor(dir)*:
12 $\quad$ **return** the peripheral neighbor in direction dir (see Section 5.2);

13 **Function** *getNeighbor(dir, pos)*:
14 $\quad$ **return** $C_k \in \mathcal{N}_{C_i}^N$ such that C_i is connected to C_k on the connected interface that immediately follows the interface pointing to position pos in direction dir;

```
15 Function requestClearance():
16     C_k ← getNeighbor(d);
17     p'_{C_i} ← position after rotation in direction d around C_k;
18     r ← (src ← p_{C_i}, dest ← p'_{C_i}, cnt ← 0);
19     send CLEARANCE_REQUEST(r) to C_k;

20 Function forwardClearance(c(src, dest), C_j):
21     if areAdjacentCells(c.src, p_{C_i}) then
22         C_k ← getNeighbor(oppositeDirection(d), c.src);
23         if C_k ≠ C_j AND areAdjacentCells(c.src, p_{C_k}) then
24             send CLEARANCE(c) to C_k;
25         else
26             Movings ← Movings ∪ {c.src};
27             send CLEARANCE(c) to C_l | p_{C_l} = c.src;
28     else if areAdjacentCells(c.dest, p_{C_i}) then
29         C_k ← getNeighbor(oppositeDirection(d), c.dest);
30         send CLEARANCE(c) to C_k;

31 Function forwardEndOfMove(c(src, dest), C_j):
32     if areAdjacentCells(c.src, p_{C_i}) then
33         C_k ← getNeighbor(oppositeDirection(d), c.src);
34         if C_k ≠ C_j AND areAdjacentCells(c.src, p_{C_k}) then
35             send END_OF_MOVE(c) to C_k;
36     else if areAdjacentCells(c.dest, p_{C_i}) then
37         C_k ← getNeighbor(oppositeDirection(d), c.dest);
38         send END_OF_MOVE(c) to C_k;
```

Algorithm 15: C2SR helper functions detailed for any module C_i.

```
1  When CLEARANCE_REQUEST(r(src, dest, cnt)) is received by C_i from C_j do:
2  if state = WAITING then
3      send DELAYED_CLEARANCE(r) to C_j;
4      return;
5  if r.dest ∈ Movings then
6      Pendings ← Pendings ∪ {r};
7      return;
8  if state = BLOCKED OR state = GOAL then
9      if r.cnt = 3 then
10         send DELAYED_REQUEST(r) to C_j;
11         return;
12     r.cnt ← r.cnt + 1;
13 C_n ← getNeighbor(d, r.dest);
14 if C_n ≠ C_j AND areAdjacentCells(p_{C_n}, r.dest) then
15     send CLEARANCE_REQUEST(r) to C_n;
16 else
17     c ← (r.src, r.dest);
18     Movings ← Movings ∪ {r.dest};
19     forwardClearance(c, ⊥);
```

```
20 When CLEARANCE(c(src, dest)) is received by C_i from C_j do:
21 if c.src = p_{C_i} then
22     clearance ← c;
23     send START_TO_MOVE to C_j;
24 else
25     forwardClearance(c, C_j);

26 When DELAYED_CLEARANCE(r(src, dest, cnt)) is received by C_i from C_j do:
27 if r.src ≠ p_{C_i} then
28     Pendings ← Pendings ∪ {r};

29 When START_TO_MOVE is received by C_i from C_j do:
30 send START_TO_MOVE_ACK to C_j;

31 When START_TO_MOVE_ACK is received by C_i from C_j do:
32 state ← MOVING;
33 C_k ← getNeighbor(d);
34 move around C_k in direction d;

35 When END_OF_MOVE(c(src, dest)) is received by C_i from C_j do:
36 Movings ← Movings − {c.src, c.dest};
37 forwardEndOfMove(c, C_j);
38 if isInStream() then
39     state ← WAITING;
40     requestClearance();
41 else if ∃r ∈ Pendings | r ∈ areAdjacentCells(r.dest, c.src) then
42     C_n ← getNeighbor(d, r.dest);
43     if areAdjacentCells(r.dest, p_{C_n}) then
44         send CLEARANCE_REQUEST(r) to C_n;
45     else
46         cl ← (r.src, r.dest);
47         Movings ← Movings ∪ {cl.dest};
48         forwardClearance(cl, ⊥);
```

Algorithm 16: C2SR algorithm message handler detailed for any module C_i.

```
1 When C_i has finished to move do:
2 p_{C_i} ← clearance.dest;
3 send END_OF_MOVE(clearance) to getNeighbor(d);
4 clearance ←⊥;
5 if hasConverged() then
6     state ← GOAL;
7 else
8     state ← WAITING;
9     requestClearance();
```

Algorithm 17: C2SR algorithm event handler detailed for any module C_i.

stream progression rule (Rule 3) is satisfied. WAITING modules send CLEARANCE_REQUEST messages to get the authorization to move. Clearance requests are composed of the module source position and of its destination. These requests travel around the module destination cell. At each hop, modules check if the requested move satisfies the stream progression rule (see Algorithm 16, lines 1-19). If the stream progression rule is not satisfied, the clearance request has either to be stored locally (see Algorithm 16, lines

5-7) or to be stored at the previous module using a DELAYED_CLEARANCE message (see Algorithm 16, lines 2-4, 9-11 and 26-28). If the stream progression rule is satisfied, the clearance is granted (see Algorithm 16, lines 16-19). The clearance is then progressively forwarded back to the module that has initiated the request (see Algorithm 16, lines 20-25).

To prevent collision, modules maintain a list of neighbor cells from/into which a module is moving. After having moved to a new position, modules send an END_OF_MOVE (EOM for short) message that is progressively forwarded around the cell of their previous position (see Algorithm 17, line 3 and Algorithm 16, lines 35-48). Upon reception of an EOM message, delayed clearances are potentially re-activated (see Algorithm 16, lines 41-48).

START_TO_MOVE and START_TO_MOVE_ACK messages guarantee that no message is lost when a module decides to actually move (see Algorithm 16, lines 29-34).

Modules never need to communicate with modules farther than two cells away in the lattice, which means that, due to our requirements, modules never need to send messages that have to travel more than five hops. Thus, our algorithm uses only local interactions between modules.

5.6/ EXPERIMENTAL EVALUATION

We implemented C2SR and evaluated it using VisibleSim, our simulator for modular robotic systems. This section presents our experimental results. Through our experiments, we show the effectiveness of C2SR and its efficiency in terms of communications, movements and execution time.

VisibleSim enables one to perform simulations with different and variable motion and communication delays. In our evaluation, we assume that neighboring modules communicate together using 8-N-1 serial communications. Hence, we assume that the effective bitrate is equal to 80% of the link bitrate. We assume that the effective average communication bitrate between two neighboring modules follows a Gaussian distribution. Moreover, we assume that the average motion speed during the atomic moves of a 2D Catom also follows a Gaussian distribution. We do not simulate delays due to processing and interruptions as we assume them to be negligible in comparison to communication and motion delays.

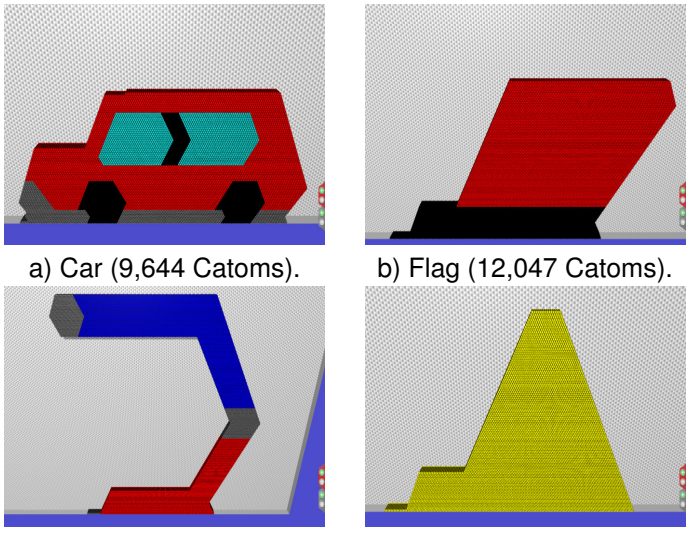
Unless explicitly mentioned, we assume the following simulation parameters. We consider that the effective average communication bitrate during message exchanges between two neighboring modules has a distribution centered on 38.9 kbit/s with a standard deviation of 389 bit/s (1% of the mean). In the current hardware prototype, a 2D Catom can move at a speed of 1.88 $mm \times s^{-1}$ (see Section 2.3.2). We assume that the average motion speed during the atomic moves of a module has a distribution centered on 1.88 $mm \times s^{-1}$ with a standard deviation of 0.0188 $mm \times s^{-1}$ (1% of the mean).

We evaluate C2SR on the self-reconfiguration of random clumps of 2D Catoms into four

kinds of shapes, namely a car, a flag, a magnet and a pyramid shape (see Figures 5.1 and 5.8). For each target shape, we generated different versions of the goal configurations using different scales ranging from a dozen to ten thousand modules. For every single point on the result plots, 10 were performed.

5.6.1/ Effectiveness Evaluation

As shown in Figure 5.8, C2SR is able to self-reconfigure ensembles composed of more than 10,000 2D Catoms.



a) Car (9,644 Catoms). b) Flag (12,047 Catoms).

c) Magnet (10,220 Catoms). d) Pyramid (8,033 Catoms).

Figure 5.8: Screenshots of VisibleSim at the end of the simulation of C2SR with different kinds of goal shapes composed of about 10,000 2D Catoms.

5.6.2/ Communication Evaluation

Figure 5.9 shows the total number of messages sent during the execution of C2SR according to the size of the goal shape. For the shapes we considered, the number of messages seems to depend on the size of the goal configuration and not on the actual shape of the arrangement. Moreover, the standard deviation is very small, so small that it is not visible in the figure. Thus, for a goal shape of a given size, C2SR always sends approximately the same number of messages. Furthermore, as shown in Figure 5.9 by the curve of best fit $y(x) = 20.29x^{1.53}$, this number of messages is highly predictable and increases polynomially with the size of the goal shape.

Figure 5.10 indicates that a few modules tend to send a lot more messages than the other modules. Intuitively, modules that stay at the boundary between $\mathcal{I}$ and $\mathcal{G}$ are communication hotspots because many modules have to communicate with them before rolling over them in order to reach $\mathcal{G}$ (see Figure 5.14).

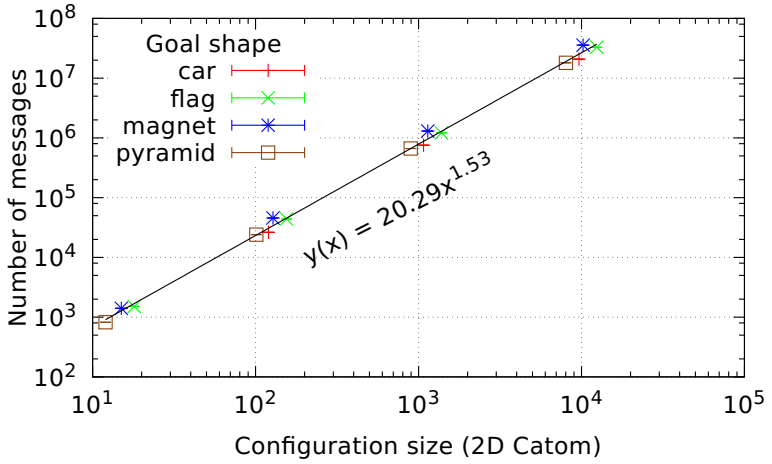


Figure 5.9: Average total number of messages (± standard deviation) sent in C2SR versus the size of the system for different goal shapes.

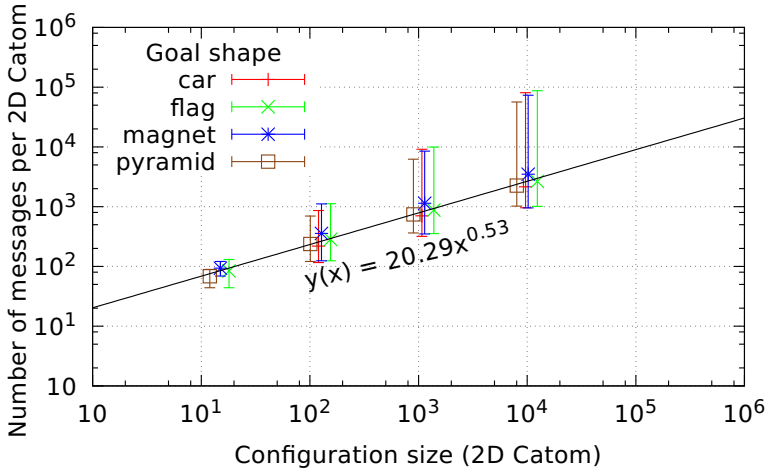


Figure 5.10: Average number of messages sent per 2D Catom (± min/-max) during the execution of C2SR versus the size of the system for different goal shapes.

Figure 5.11 shows the maximum message queue size reached by the modules during the execution of C2SR, taking into account both the incoming and the outgoing messages. The maximum message queue size is constant and equal to two, regardless of the shape of the goal configuration and regardless of its size. We recall that messages generated by C2SR have a small and constant size. As a consequence, the traffic generated by C2SR is well controlled and modules do not require a lot of memory space to store incoming and outgoing messages.

Figure 5.12 shows the average number of hops traveled by the packets during the execution of C2SR. The average and the maximum number of hops traveled by the packets is small and relatively constant, regardless of the shape of the goal configuration and regardless of its size. This confirms that C2SR only involves local interactions, as stated in the previous section.

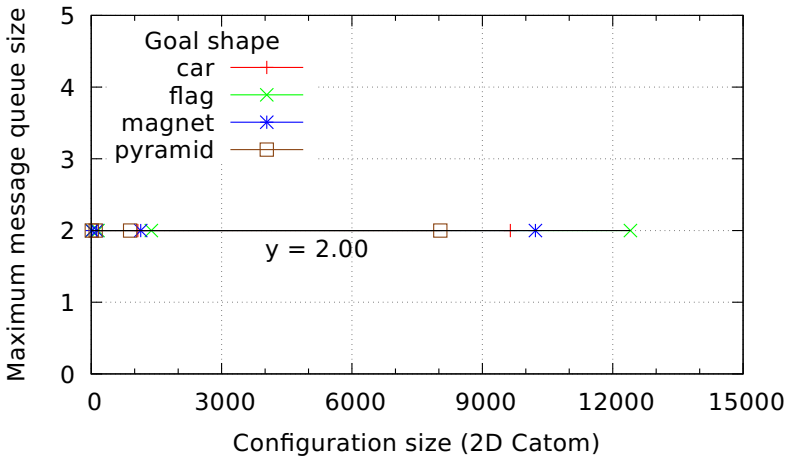


Figure 5.11: Maximum message queue size (incoming and outgoing messages) reached by any node versus the size of the system during the execution of C2SR.

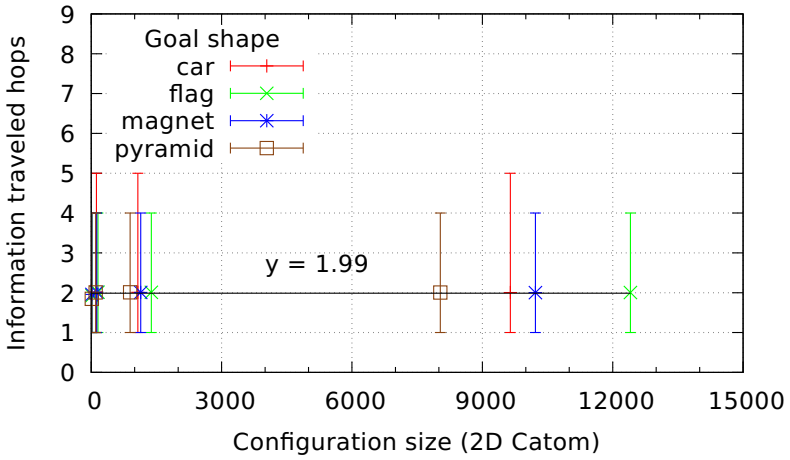


Figure 5.12: Average number of hops traveled by data (± min/max) in the execution of C2SR versus the size of the system.

5.6.3/ MOTION EFFICIENCY

Figure 5.13 shows the total number of atomic moves performed during the execution of C2SR according to the size of the system for different goal shapes. Note that this figure is really similar to Figure 5.9. Here again, the number of atomic moves seems to depend only on the size of the goal configuration and not on the actual shape of the arrangement. As shown in Figure 5.13 by the curve of best fit $y(x) = 2.09x^{1.53}$, the number of atomic moves is highly predictable and increases polynomially with the size of the goal shape. Notice that the number of messages is approximately equal to ten times the number of moves (see Figures 5.9 and 5.13). Thus, an atomic move requires on average 10 messages.

As shown in Figure 5.14, many modules can move concurrently during the execution of

C2SR. Thus, although the self-reconfiguration process may require many atomic moves, it remains reasonably time-efficient, as shown in the next subsection.

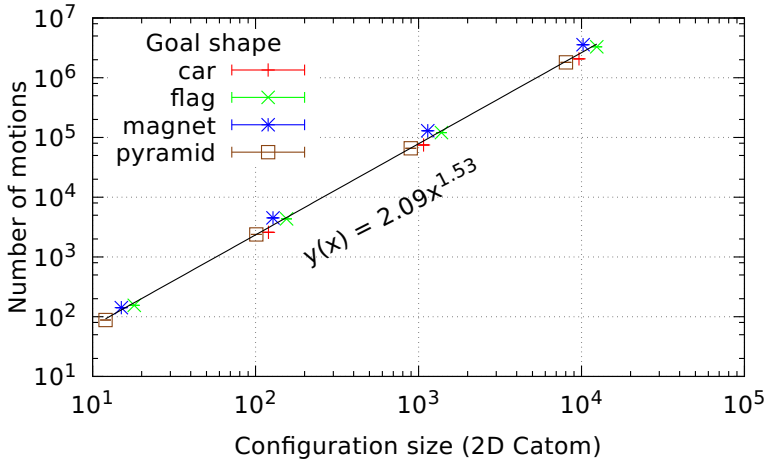


Figure 5.13: Average total number of atomic moves (± standard deviation) versus the size of the system for different goal shapes.

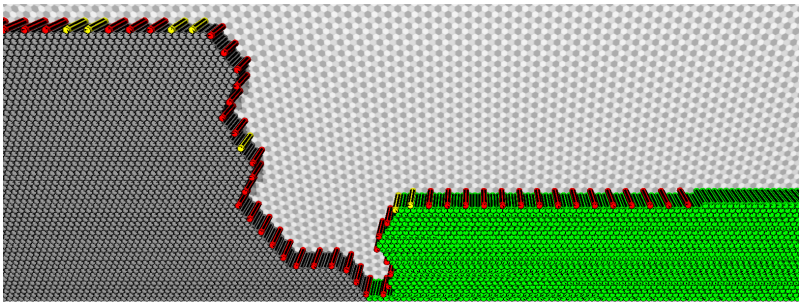


Figure 5.14: Screenshot of VisibleSim during a self-reconfiguration process with C2SR. Modules in the stream progress by rotating CW. Blocked modules are in gray, waiting ones in yellow, moving ones in red and modules that have converged are in green.

5.6.4/ Execution Time Efficiency

Figure 5.15 shows the average simulated time of C2SR execution according to the size of the system. For the different goal shapes we considered, this time seems to depend only on the size of the configuration and not on the actual shape of the arrangement. Moreover, the standard deviation is very small and not visible in the figure. Thus, for a goal shape of a given size, C2SR always approximately lasts for the same duration. As shown in Figure 5.15 by the curve of best fit $y(x) = 0.017x + 0.149$, the simulated time is highly predictable and increases linearly with the size of the goal shape. The slope of the line gives the reconfiguration speed: C2SR fills, on average, $\frac{1}{0.017} \approx 59$ goal cells per minute, i.e., approximately 1 cell per second. Note that, in these experiments, the reconfiguration speed is independent of the goal shape.

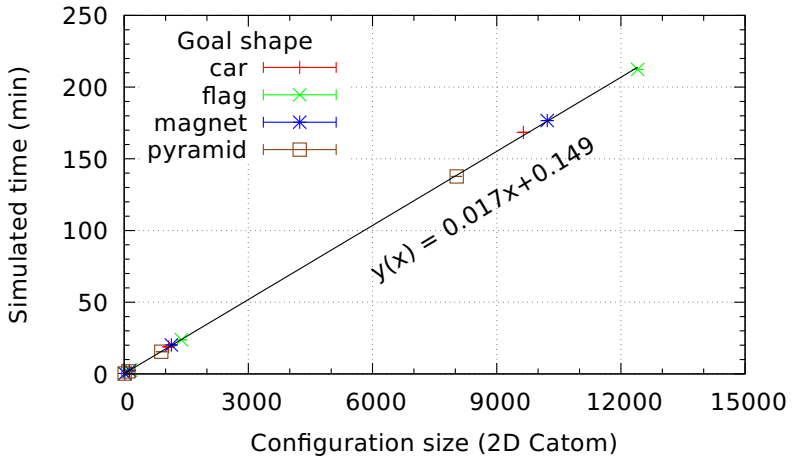


Figure 5.15: Average C2SR simulated time (± standard deviation) versus the size of the system for different goal shapes.

Figure 5.16 shows the average simulated time of the C2SR execution according to the average communication bitrate for the two different motion speeds supported by the 2D Catoms. We consider the usual bitrates of serial communications. We conducted this experiment for the car goal shape composed of 1,073 modules. Until 38.9 kbit/s, the self-reconfiguration process becomes much faster, as the average communication bitrate increases. Beyond 38.9 kbit/s, the self-reconfiguration speed increases less quickly and tends to stabilize.

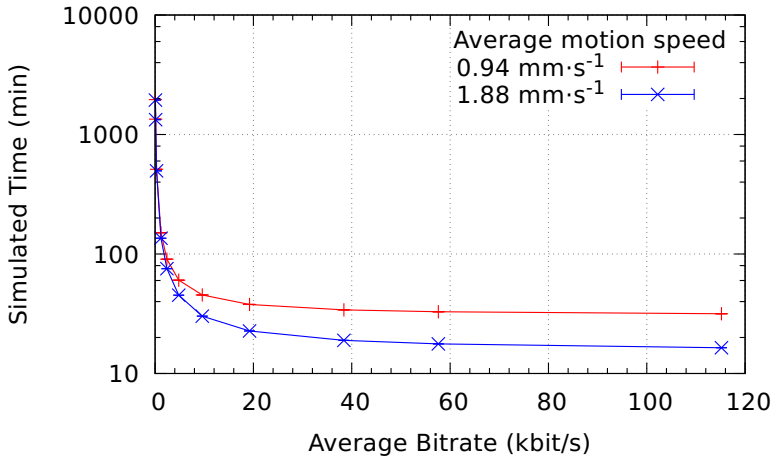


Figure 5.16: Average C2SR simulated time (± standard deviation) versus the communication bitrate (random initial configuration into the car of 1,073 2D Catoms).

5.7/ CONCLUSION

In this chapter, we proposed the Cylindrical-Catoms Self-Reconfiguration (C2SR) algorithm, a parallel, asynchronous and fully decentralized distributed algorithm to self-reconfigure a lattice-based MSR from an initial shape into a goal one. We evaluated our algorithm using simulations on ensembles with up to 10,020 Catoms. The results show that C2SR has nice properties.

6

CONCLUSION

6.1/ SUMMARY

In this work, we considered systems composed of resource-constrained modules that are organized in a lattice structure and which can only communicate with neighboring modules. We identified and implemented three high-level primitives, namely centrality-based leader election, time synchronization and self-reconfiguration.

We proposed a collection of efficient and effective distributed algorithms to elect approximate-centroid and approximate-center nodes in asynchronous distributed systems. We introduced the k-BFS SumSweep framework, the ABC-Center algorithm and the Probabilistic-Counter-based Central-Leader Election (PC2LE) framework. Our algorithms and frameworks do not require any prior knowledge of the network, have a well-defined termination criterion, converge in a reasonable amount of time and are memory-efficient. The k-BFS SumSweep framework runs in $O(kd)$ time using $O(mn^2)$ messages of size $O(1)$ and $O(\Delta)$ memory space per node. We proposed two versions of ABC-Center. The latest version, ABC-CenterV2, runs in $O(sd)$ time using $O(mn^2)$ messages of size $O(1)$ and $O(\Delta)$ memory space per node, where s is the number of iterations ABC-CenterV2 requires to terminate. PC2LE runs in $O(d)$ time using $O(mn^2)$ messages of size $O(c)$ and $O(\Delta+c)$ memory space per node, where c is the memory usage of the probabilistic counter used in PC2LE. If we consider that the maximum number of neighbors a node can have is bounded by a constant, the memory usage of our algorithms is further reduced to $O(1)$, for k-BFS SumSweep and ABC-CenterV2, and to $O(c)$ for PC2LE. It is, for instance, the case in many modular robotic systems that use neighbor-to-neighbor communications (e.g, the Blinky Blocks, the Smart Blocks, etc.). We evaluated the proposed algorithms on the Blinky Blocks modular robotic system both on hardware prototypes and through simulations. Our algorithms scale well in terms of accuracy, execution time, number of messages and memory usage. In large-scale systems with 25,000 modules, our algorithms provide a relative centroid accuracy between 96%-99% and a relative center accuracy between 88%-94%. As a consequence, our algorithms are suitable for large-scale embedded distributed systems with scarce memory, computing and energy resources. To the best of our knowledge, our algorithms are the most precise existing distributed algorithms designed to elect an approximate centroid or an approximate center in our target systems, with both a reasonable convergence time and a limited storage cost.

Furthermore, we introduced the Modular Robot Time Protocol (MRTP), a network-wide time synchronization protocol for modular robots. Our protocol achieves its performance by combining several mechanisms: central time master election, fast and recursive propagation of synchronization waves along the edges of a breadth-first spanning tree, low-level timestamping with per-hop compensation for communication delays using the most-appropriate method for the target platform, and clock skew compensation using linear regression. We evaluated our protocol on the Blinky Blocks system both on hardware and through simulations. Experimental results show that MRTP can potentially manage real systems composed of up to 27,775 Blinky Blocks. Furthermore, we showed that our protocol is able to keep a Blinky Blocks system synchronized to a few milliseconds, using few network resources at runtime, even though the Blinky Blocks use low-bitrate communications (38.4 kbit/s) and are equipped with very low-accuracy (10,000 parts per million (ppm)) and poor-resolution (1 millisecond) clocks. We compared MRTP to existing synchronization protocols ported to fit our system model. Simulation results show that MRTP can achieve better synchronization precision than the most precise protocols compared, while sending more than half less messages in compact systems.

Additionally, we presented the Cylindrical-Catoms Self-Reconfiguration (C2SR) algorithm, a self-reconfiguration algorithm for rolling cylindrical modules arranged in a two-dimensional vertical hexagonal lattice. Our algorithm is a parallel, asynchronous and decentralized distributed algorithm allowing the self-reconfiguration of robots from an initial configuration into a goal one. It is able to manage almost any kind of initial and goal compact shapes (i.e., without any hole). We showed the effectiveness of our algorithm and studied its performance in terms of communications, movements and execution time using simulations. Our observations indicate that the number of communications, the number of movements and the execution time of our algorithm are highly predictable. Furthermore, we observed execution times that are linear in the size of the goal shape.

6.2/ FUTURE WORK

This section presents perspectives on future research. We first discuss improvements to our three primitives and then suggest more general future work.

Centrality-based Leader Election In future work, it will be interesting to carry out a formal analysis of the accuracy of our algorithms in order to derive bounds or to try to find bad cases, where our algorithms fail. For now, we did not faced any really bad case during our experiments.

PC2LE estimates d, the diameter of the network, to bound the number of rounds. With the method proposed in this chapter, the estimation is upper-bounded by $2d$. Thus, in the worst case, PC2LE unnecessarily performs d rounds, which uses $O(d)$ time and generates $O(dm)$ messages for nothing. In future work, it will be interesting to find an efficient method to better estimate d.

Furthermore, our work on network centrality can potentially be applied to a wide variety of

distributed systems. In future work, we plan to evaluate the performance of our algorithms on different systems and, if necessary, to propose system-specific adaptations.

The number of iterations required for ABC-Center to terminate increases with the diameter thickness. Intuitively, the number of iterations tends to increase with the network density, as the number of equidistant nodes between any two nodes tends to be greater in dense networks. While ABC-Center requires only a few iterations in modular robotic systems where nodes are organized in a simple-cubic lattice, its efficiency has to be studied in other types of networks.

In addition, we plan to study the problems of centrality-based leader election in networks that exhibit a high degree of dynamics due to nodes failure and/or mobility. Currently, our algorithms restart computations from scratch upon neighbor change detection. This mechanism will be too expensive in terms of resource usage in highly dynamic networks.

We also plan to extend the controlled-broadcast optimization, proposed in Section 3.5.1.2, into a framework that will make it possible to run multiple BFS traversals (including election traversals) in parallel, without network congestion. In the envisaged framework, we will ensure that at most $O(1)$ BFS messages for all BFS traversals will be present in any outgoing-message queue at all times. We would like to use this framework to design a more efficient version of the k-BFS-RAND-PAR algorithm (see Section 3.9.3.1).

Time Synchronization We plan to test MRTP in large-scale hardware systems running real applications, which have time synchronization requirements and which may potentially generate a significant network and computing load.

In addition, it would be interesting to design more precise methods of compensating for network delays in Blinky Blocks systems. We envision, for instance, to enhance FD with a method that will compensate for the dissemination error after several hops, i.e., when this error has become greater than the resolution of the clock and can effectively be compensated for. Also, different network delay compensation methods can be combined to provide a better estimation of the current global time. In order not to increase the communication load, a same message can carry multiple timestamps inserted by different methods.

In future versions of MRTP, we want to consider both centrality and clock stability in the time-master election. We also want to adapt MRTP to deal with outlier slave modules equipped with less stable clocks than the others.

Furthermore, MRTP should be tested in other modular robotic systems that fit its system model. In particular, it will be interesting to determine if the predictive method to compensate for communication delays is still more precise than the other methods in systems with higher hardware-clock accuracy.

Moreover, we plan to study time synchronization in highly dynamic modular robotic systems where module mobility and failures may occur frequently. In particular, we want to address the problem of time synchronization throughout the process of self-reconfiguration, during which modules move to rearrange the global shape of the modular

robot (e.g., [Piranda et al., 2016a],[Lakhlef et al., 2014]). MRTP needs to be adapted to efficiently handle such network dynamics, because the frequent re-elections of a central module and the maintenance of the synchronization tree will be too expensive. For now, we suggest using the high-level framework of the PulseSync protocol [Lenzen et al., 2015] in those systems. This framework is indeed inherently tolerant of module mobility and failures.

Self-Reconfiguration In future work, we will demonstrate the correctness of C2SR, i.e., we will prove that the goal configuration can be built if the shape admissibility conditions are satisfied. Moreover, we will study the performance of C2SR on other types of shapes and compare it to existing algorithms. We will also study the distribution of both the number of messages sent per module and the number of atomic moves performed per module. Our observations seem to indicate that our algorithm is highly predictable and that its execution time is linear with respect to the size of the goal shape. A further step would be to prove it.

In the design of C2SR, we tried to prevent modules from unnecessarily climbing over others, assuming that going up may consume more energy. C2SR fulfills this goal when modules travel on the periphery of the initial shape, which is progressively stripped so that no module can go up. However, if the goal shape contains hills on its periphery, hills close to the initial shape will be completely constructed before modules can continue to roll on the periphery of the goal shape. Hence, many modules will then have to climb up these hills to reach the other side of the goal shape. We would like to overcome this limitation in the future version of C2SR. Peripheral modules that have already converged can, for instance, advertise remote modules in the stream about farther goal cells to be filled, thus, causing modules in the stream not to freeze in a hill.

Modules of modular robotic ensembles are low-cost mass-produced tiny electronic devices that are inherently prone to failures. Failures should then be considered when designing primitives for these ensembles. In particular, we do not consider module failures in our self-reconfiguration algorithm and it would be interesting to adapt our algorithm so that it can cope with such failures.

In addition, it will be interesting to extend our algorithm so that it will be able to cope with 3D modular robotic systems such as the 3D Catoms [Piranda et al., 2016b] which can roll over neighboring modules in the 3D space. It could be done by constructing the goal shape plane by plane, every plane being constructed line by line, as in the current version of C2SR. However, in ensembles of 3D Catoms, several paths may exist to reach a given cell, thus, faster approaches which allow many modules to move concurrently can be envisioned.

Set of Primitives Other primitives have to be identified and studied in future work. Some challenging algorithmic problems in large-scale robotic ensembles have already been studied for years, e.g., robot localization [Funiak et al., 2009, Moffo et al., 2016], reconfiguration goal shape compression [Tucci et al., 2017], locomotion [Fitch et al., 2007],

coating [Derakhshandeh, 2017], reconfiguration termination detection [Butler et al., 2002]. In my opinion, other interesting primitives include data dissemination, data sharing, message routing and construction of a virtual representation of robot ensembles.

In chapter 2, we explain how Programmable Matter (PM) could be used to enhance the computer-aided design process because PM provides a consistent mapping between the virtual and physical representations of a same object. Hardware modules have limited memory capacity and may not afford to store the complete reconfiguration goal shape even in a compressed format. Data dissemination algorithms, data sharing protocols combined with appropriate routing methods will, for instance, enable to disseminate and share large virtual representations between all modules. In this approach, every module does not store the complete virtual representation of an object but instead, only a part of it and can transparently access locally and remotely stored parts of the goal representation. This will reduce the individual memory usage of modules during the reconfiguration process.

The construction of a virtual representation of robot ensembles would enable designers to update their virtual representation of an object after having manually modified its physical representation made of PM. The virtual representation could possibly be reconstructed using external means (e.g., cameras and imagery processing) or by the modules themselves using communications.

Furthermore, it will be interesting to release primitive implementations in a set of libraries in order to provide a complete software environment for large-scale distributed modular robotic ensemble coordination.

PUBLICATIONS

André Naz, Benoît Piranda, Julien Bourgeois, and Seth Copen Goldstein. **A Time Synchronization Protocol for Large-Scale Distributed Embedded Systems with Low-Precision Clocks and Neighbor-to-Neighbor Communications.** *In Journal of Network and Computer Applications (JNCA),* accepted on December 2017. Elsevier.

André Naz, Benoît Piranda, Thadeu Tucci, Seth Copen Goldstein, and Julien Bourgeois. **Network Characterization of Lattice-based Modular Robots with Neighbor-to-Neighbor Communications.** *In 2016 13th International Symposium on Distributed Autonomous Robotic Systems (DARS),* pages 415 – 429, London, UK, November 2016. Springer..

André Naz, Benoît Piranda, Seth Copen Goldstein, and Julien Bourgeois. **A Time Synchronization Protocol for Modular Robots.** *In PDP 2016, 24th Euromicro Int. Conf. on Parallel, Distributed, and Network-Based Processing,* pages 109 – 118, Heraklion Crete, Greece, February 2016. IEEE. Core Rank: C. Acceptance Rate: 32%.

André Naz, Benoît Piranda, Seth Copen Goldstein, and Julien Bourgeois. **A Distributed Self-Reconfiguration Algorithm for Cylindrical Lattice-based Modular Robots.** *In 2016 IEEE 15th International Symposium on Network Computing and Applications (NCA),* pages 254 – 263, Cambridge, MA, USA, November 2016. IEEE. Core Rank: A. Acceptance Rate: 28%.

André Naz, Benoît Piranda, Seth Copen Goldstein, and Julien Bourgeois. **Approximate-Centroid Election in Large-Scale Distributed Embedded Systems.** *In AINA 2016, 30th IEEE International Conference on Advanced Information Networking and Applications,* pages 548 – 556, Crans-Montana, Switzerland, March 2016. IEEE. Core Rank: B. Acceptance Rate: 29%.

Julien Bourgeois, Benoît Piranda, André Naz, Nicolas Boillot, Hakim Mabed, Dominique Dhoutaut, Thadeu Tucci, and Hicham Lakhlef. **Programmable Matter as a Cyber-Physical Conjugation.** *In 2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC 2016),* pages 2942 – 2947, Budapest, Hungary, October 2016. IEEE. Core Rank: B.

André Naz, Benoît Piranda, Seth Copen Goldstein, and Julien Bourgeois. **ABC-Center: Approximate-Center Election in Modular Robots.** *In 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS),* pages 2951 – 2957, Hamburg, Germany, September 2015. Core Rank: A. Acceptance Rate: 46%.

BIBLIOGRAPHY

[Ahmadzadeh et al., 2016] Ahmadzadeh, H., Masehian, E., and Asadpour, M. (2016). **Modular robotic systems: Characteristics and applications**. *J. Intell. Robotics Syst.*, 81(3-4):317–357.

[Albert et al., 1999] Albert, R., Jeong, H., and Barabási, A.-L. (1999). **Internet: Diameter of the world-wide web**. *Nature*, 401(6749):130–131.

[Allan, 1987] Allan, D. W. (1987). **Time and frequency (time-domain) characterization, estimation, and prediction of precision clocks and oscillators**. *IEEE transactions on ultrasonics, ferroelectrics, and frequency control*, 34(6):647–654.

[Almeida et al., 2012] Almeida, P. S., Baquero, C., and Cunha, A. (2012). **Fast distributed computation of distances in networks**. In *Decision and Control (CDC), 2012 IEEE 51st Annual Conference on*, pages 5215–5220. IEEE.

[Amundson et al., 2008] Amundson, I., Kusy, B., Volgyesi, P., Koutsoukos, X., and Ledeczi, A. (2008). **Time synchronization in heterogeneous sensor networks**. In *Distributed Computing in Sensor Systems*, pages 17–31. Springer.

[Appleby, 2011] Appleby, A. (2011). **Murmur3 hash function**. https://github.com/aappleby/smhasher.

[Aspnes, 2017] Aspnes, J. (2017). **Notes on theory of distributed systems cpsc 465/565: Fall 2017**.

[ATMEL, 2013] ATMEL (2013). **XMEGA A3 microcontroller data-sheet**.

[ATMEL, 2016] ATMEL (2016). **AVR1003: using the XMEGA™ clock system**.

[Awerbuch, 1985] Awerbuch, B. (1985). **Complexity of network synchronization**. *Journal of the ACM (JACM)*, 32(4):804–823.

[Awerbuch et al., 1985] Awerbuch, B., and Gallager, R. G. (1985). **Distributed bfs algorithms**. In *Foundations of Computer Science, 1985, 26th Annual Symposium on*, pages 250–256. IEEE.

[Baca et al., 2010] Baca, J., Ferre, M., Collar, M., Fernandez, J., and Aracil, R. (2010). **Synchronizing a modular robot colony for cooperative tasks based on intra-inter robot communications**. In *Electronics, Robotics and Automotive Mechanics Conference (CERMA), 2010*, pages 388–393. IEEE.

[Barraquand et al., 1991] Barraquand, J., and Latombe, J.-C. (1991). **Robot motion planning: A distributed representation approach**. *The International Journal of Robotics Research*, 10(6):628–649.

[Barrenetxea et al., 2006] Barrenetxea, G., Berefull-Lozano, B., and Vetterli, M. (2006). **Lattice networks: Capacity limits, optimal routing, and queueing behavior**. *IEEE/ACM Transactions on Networking (TON)*, 14(3):492–505.

[Barthélemy, 2011] Barthélemy, M. (2011). **Spatial networks**. *Physics Reports*, 499(1):1–101.

[Bateau et al., 2012] Bateau, J., Clark, A., McEachern, K., Schutze, E., and Walter, J. (2012). **Increasing the efficiency of distributed goal-filling algorithms for self-reconfigurable hexagonal metamorphic robots**. In *Proceedings of the International Conference on Parallel and Distributed Techniques and Applications*, pages 509–515.

[Bhalla et al., 2007] Bhalla, N., and Jacob, C. (2007). **A framework for analyzing and creating self-assembling systems**. In *2007 IEEE Swarm Intelligence Symposium*, pages 281–288. IEEE.

[Blazevic et al., 2005] Blazevic, L., Le Boudec, J.-Y., and Giordano, S. (2005). **A location-based routing method for mobile ad hoc networks**. *IEEE Transactions on mobile computing*, 4(2):97–110.

[Bonacich, 1972] Bonacich, P. (1972). **Factoring and weighting approaches to status scores and clique identification**. *Journal of Mathematical Sociology*, 2(1):113–120.

[Borassi et al., 2014] Borassi, M., Crescenzi, P., Habib, M., Kosters, W., Marino, A., and Takes, F. (2014). **On the solvability of the six degrees of kevin bacon game**. In *Fun with Algorithms*, pages 52–63. Springer.

[Boulinier et al., 2008] Boulinier, C., Datta, A. K., Larmore, L. L., and Petit, F. (2008). **Space efficient and time optimal distributed bfs tree construction**. *Information Processing Letters*, 108(5):273–278.

[Bourgeois et al., 2012] Bourgeois, J., and Goldstein, S. (2012). **Distributed intelligent mems: Progresses and perspectives**. *ICT Innovations 2011*, pages 15–25.

[Bourgeois et al., 2016] Bourgeois, J., Piranda, B., Naz, A., Lakhlef, H., Boillot, N., Mabed, H., Douthaut, D., and Tucci, T. (2016). **Programmable matter as a cyber-physical conjugation**. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, pages 2942–2947, Budapest, Hungary. IEEE.

[Braden, 1989] Braden, R. (1989). **Requirements for Internet Hosts – Communication Layers**. RFC 1122, RFC Editor.

[Bruell et al., 1999] Bruell, S. C., Ghosh, S., Karaata, M. H., and Pemmaraju, S. V. (1999). **Self-stabilizing algorithms for finding centers and medians of trees**. *SIAM Journal on Computing*, 29(2):600–614.

[Butler et al., 2002] Butler, Z., Fitch, R., Rus, D., and Wang, Y. (2002). **Distributed goal recognition algorithms for modular robots**. In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, volume 1, pages 110–116. IEEE.

[Cardozo et al., 2012] Cardozo, T. B., Silva, A. P. C., Vieira, A. B., and Ziviani, A. (2012). **On the end-to-end connectivity evolution of the internet**.

[Chan et al., 2009] Chan, S. Y., Leung, I. X., and Liò, P. (2009). **Fast centrality approximation in modular networks**. In *Proceedings of the 1st ACM international workshop on Complex networks meet information & knowledge management*, pages 31–38. ACM.

[Chechik et al., 2014] Chechik, S., Larkin, D. H., Roditty, L., Schoenebeck, G., Tarjan, R. E., and Williams, V. V. (2014). **Better approximation algorithms for the graph diameter**. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 1041–1052. SIAM.

[Chennareddy et al., 2017] Chennareddy, S., Agrawal, A., and Karuppiah, A. (2017). **Modular self-reconfigurable robotic systems: A survey on hardware architectures**. *Journal of Robotics*, 2017.

[Chepoi et al., 1994] Chepoi, V., and Dragan, F. (1994). **A linear-time algorithm for finding a central vertex of a chordal graph**. *Algorithms—ESA'94*, pages 159–170.

[Cheung, 1983] Cheung, T.-Y. (1983). **Graph traversal techniques and the maximum flow problem in distributed computation**. *Software Engineering, IEEE Transactions on*, SE-9(4):504–512.

[Crescenzi et al., 2013] Crescenzi, P., Grossi, R., Habib, M., Lanzi, L., and Marino, A. (2013). **On computing the diameter of real-world undirected graphs**. *Theoretical Computer Science*, 514:84–95.

[Cristian, 1989] Cristian, F. (1989). **Probabilistic clock synchronization**. *Distributed Computing*, 3(3):146–158.

[De Rosa et al., 2006] De Rosa, M., Goldstein, S., Lee, P., Campbell, J., and Pillai, P. (2006). **Scalable shape sculpting via hole motion: Motion planning in lattice-constrained modular robots**. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, ICRA 2006.*, pages 1462–1468. IEEE.

[Derakhshandeh, 2017] Derakhshandeh, Z. (2017). **Algorithmic Foundations of Self-Organizing Programmable Matter**. PhD thesis, Arizona State University.

[Derakhshandeh et al., 2014] Derakhshandeh, Z., Dolev, S., Gmyr, R., Richa, A. W., Scheideler, C., and Strothmann, T. (2014). **Brief announcement: amoebot–a new model for programmable matter**. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, pages 220–222. ACM.

[Dhoutaut et al., 2013] Dhoutaut, D., Piranda, B., and Bourgeois, J. (2013). **Efficient simulation of distributed sensing and control environments**. In *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*, pages 452–459. IEEE.

[Dissler et al., 2016] Dissler, B., Holzer, S., and Wattenhofer, R. (2016). **Distributed local multi-aggregation and centrality approximation**. *arXiv preprint arXiv:1605.06882*.

[Dutot et al., 2011] Dutot, A., Olivier, D., and Savin, G. (2011). **Centroids : a decentralized approach**. In *ECCS - European Conference on Complex Systems*, Vienna, Austria.

[Eicken et al., 1992] Eicken, T., Culler, D. E., Goldstein, S. C., and Schauser, K. E. (1992). **Active messages: a mechanism for integrated communication and computation**. In *Computer Architecture, 1992. Proceedings of the 19th Annual International Symposium on*, pages 256–266. IEEE.

[Ellis et al., 2004] Ellis, R. B., Martin, J. L., and Yan, C. (2004). **Random geometric graph diameter in the unit disk with ℓ p metric**. In *International Symposium on Graph Drawing*, pages 167–172. Springer.

[Elson et al., 2002] Elson, J., Girod, L., and Estrin, D. (2002). **Fine-grained network time synchronization using reference broadcasts**. *ACM SIGOPS Operating Systems Review*, 36(SI):147–163.

[Eppstein et al., 2001] Eppstein, D., and Wang, J. (2001). **Fast approximation of centrality**. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 228–229. Society for Industrial and Applied Mathematics.

[Etzlinger et al., 2014] Etzlinger, B., Wymeersch, H., and Springer, A. (2014). **Cooperative synchronization in wireless networks**. *IEEE Transactions on Signal Processing*, 62(11):2837–2849.

[Fekete et al., 2016] Fekete, S., Richa, A. W., Römer, K., and Scheideler, C. (2016). **Algorithmic Foundations of Programmable Matter (Dagstuhl Seminar 16271)**. *Dagstuhl Reports*, 6(7):1–14.

[Ferrari et al., 2011] Ferrari, F., Zimmerling, M., Thiele, L., and Saukh, O. (2011). **Efficient network flooding and time synchronization with Glossy**. In *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*, pages 73–84. IEEE.

[Fitch et al., 2007] Fitch, R., and Butler, Z. (2007). **Scalable locomotion for large self-reconfiguring robots**. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 2248–2253. IEEE.

[Flajolet et al., 2007] Flajolet, P., Fusy, É., Gandouet, O., and Meunier, F. (2007). **Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm**. In *AofA: Analysis of Algorithms*, pages 137–156. Discrete Mathematics and Theoretical Computer Science.

[Flajolet et al., 1985] Flajolet, P., and Nigel Martin, G. (1985). **Probabilistic counting algorithms for data base applications**. *Journal of computer and system sciences*, 31(2):182–209.

[Flury et al., 2010] Flury, R., and Wattenhofer, R. (2010). **Slotted programming for sensor networks**. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 24–34. ACM.

[Fowler et al., 1991] Fowler, G., Noll, L. C., Vo, K.-P., and Eastlake, D. (1991). **The FNV non-cryptographic hash algorithm**. http://www.isthe.com/chongo/tech/comp/fnv/index.html.

[Freeman et al., 1979] Freeman, L. C., Roeder, D., and Mulholland, R. R. (1979). **Centrality in social networks: ii. experimental results**. *Social networks*, 2(2):119–141.

[Funiak et al., 2009] Funiak, S., Pillai, P., Ashley-Rollman, M. P., Campbell, J. D., and Goldstein, S. C. (2009). **Distributed localization of modular robot ensembles**. *International Journal of Robotics Research*, 28(8):946–961.

[Gallager, 1982] Gallager, R. G. (1982). **Distributed minimum hop algorithms**. Technical Report, MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR INFORMATION AND DECISION SYSTEMS.

[Ganeriwal et al., 2003] Ganeriwal, S., Kumar, R., and Srivastava, M. B. (2003). **Timing-sync protocol for sensor networks**. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 138–149. ACM.

[Garcia et al., 2009] Garcia, R. F. M., Schultz, U. P., and Stoy, K. (2009). **On the efficiency of local and global communication in modular robots**. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 1502–1508. IEEE.

[Garin et al., 2012] Garin, F., Varagnolo, D., and Johansson, K. H. (2012). **Distributed estimation of diameter, radius and eccentricities in anonymous networks**. *IFAC Proceedings Volumes*, 45(26):13–18.

[Gibbons, 2016] Gibbons, P. B. (2016). **Distinct-values estimation over data streams**. In *Data Stream Management*, pages 121–147. Springer.

[Gilpin et al., 2010] Gilpin, K., Knaian, A., and Rus, D. (2010). **Robot pebbles: One centimeter modules for programmable matter through self-disassembly.** In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2485–2492.

[Gilpin et al., 2008] Gilpin, K., Kotay, K., Rus, D., and Vasilescu, I. (2008). **Miche: Modular shape formation by self-disassembly**. *The International Journal of Robotics Research*, 27(3-4):345–372.

[Goldstein et al., 2004] Goldstein, S. C., and Mowry, T. C. (2004). **Claytronics: An instance of programmable matter**. In *Wild and Crazy Ideas Session of ASPLOS*, Boston, MA.

[Gusella et al., 1989] Gusella, R., and Zatt, S. (1989). **The accuracy of the clock synchronization achieved by tempo in berkeley unix 4.3 bsd**. *Software Engineering, IEEE Transactions on*, 15(7):847–853.

[Handler, 1973] Handler, G. Y. (1973). **Minimax location of a facility in an undirected tree graph**. *Transportation Science*, 7(3):287–293.

[Hanneman et al., 2005] Hanneman, R. A., and Riddle, M. (2005). **Introduction to social network methods**.

[Hawkes et al., 2010] Hawkes, E., An, B., Benbernou, N. M., Tanaka, H., Kim, S., Demaine, E. D., Rus, D., and Wood, R. J. (2010). **Programmable matter by folding**. *Proceedings of the National Academy of Sciences*, 107(28):12441–12445.

[Hayes, 2000] Hayes, B. (2000). **Graph theory in practice: Part ii**. *American Scientist*, 88(2):104–109.

[He et al., 2014a] He, J., Cheng, P., Shi, L., Chen, J., and Sun, Y. (2014a). **Time synchronization in wsns: A maximum-value-based consensus approach**. *IEEE Transactions on Automatic Control*, 59(3):660–675.

[He et al., 2014b] He, J., Li, H., Chen, J., and Cheng, P. (2014b). **Study of consensus-based time synchronization in wireless sensor networks**. *ISA transactions*, 53(2):347–357.

[Holzer et al., 2012] Holzer, S., and Wattenhofer, R. (2012). **Optimal distributed all pairs shortest paths and applications**. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, pages 355–364. ACM.

[Hou et al., 2014] Hou, F., and Shen, W.-M. (2014). **Graph-based optimal reconfiguration planning for self-reconfigurable robots**. *Robotics and Autonomous Systems*, 62(7):1047 – 1059.

[Hurtado et al., 2013] Hurtado, F., Molina, E., Ramaswami, S., and Sacristán, V. (2013). **Distributed universal reconfiguration of 2D lattice-based modular robots**. In *Proc. 29th European Workshop, Computational Geometry*, volume 139, page 142.

[IEEE, 2008] IEEE (2008). **IEEE 1588-2008: Standard for a precision clock synchronization protocol for networked measurement and control systems**. Technical Report, IEEE.

[Jennings et al., 2002] Jennings, E. H., and Okino, C. M. (2002). **On the diameter of sensor networks**. In *Aerospace Conference Proceedings, 2002. IEEE*, volume 3, pages 3–1211. IEEE.

[Jin et al., 2006] Jin, S., and Bestavros, A. (2006). **Small-world characteristics of internet topologies and implications on multicast scaling**. *Computer Networks*, 50(5):648–666.

[Kang et al., 2011a] Kang, U., Papadimitriou, S., Sun, J., and Tong, H. (2011a). **Centralities in large networks: Algorithms and observations.** In *SDM*, volume 2011, pages 119–130. SIAM.

[Kang et al., 2011b] Kang, U., Tsourakakis, C. E., Appel, A. P., Faloutsos, C., and Leskovec, J. (2011b). **Hadi: Mining radii of large graphs**. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 5(2):8.

[Karagozler, 2012] Karagozler, M. E. (2012). **Design, Fabrication and Characterization of an Autonomous, Sub-millimeter Scale Modular Robot**. PhD thesis, Carnegie Mellon University.

[Karagozler et al., 2009] Karagozler, M. E., Goldstein, S. C., and Reid, J. R. (2009). **Stress-driven mems assembly + electrostatic forces = 1mm diameter robot**. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS '09)*.

[Ke et al., 2012] Ke, Y., Ong, L. L., Shih, W. M., and Yin, P. (2012). **Three-dimensional structures self-assembled from dna bricks**. *science*, 338(6111):1177–1183.

[Kermarrec et al., 2011] Kermarrec, A.-M., Le Merrer, E., Sericola, B., and Trédan, G. (2011). **Second order centrality: Distributed assessment of nodes criticity in complex networks**. *Computer Communications*, 34(5):619–628.

[Kim et al., 2013] Kim, C., and Wu, M. (2013). **Leader election on tree-based centrality in ad hoc networks**. *Telecommunication Systems*, 52(2):661–670.

[Kim et al., 2012] Kim, H., Ma, X., and Hamilton, B. R. (2012). **Tracking low-precision clocks with time-varying drifts using kalman filtering**. *IEEE/ACM Transactions on Networking (TON)*, 20(1):257–270.

[Kim et al., 2011] Kim, J.-W., Kim, J.-H., and Deaton, R. (2011). **Dna-linked nanoparticle building blocks for programmable matter**. *Angewandte Chemie International Edition*, 50(39):9185–9190.

[Kim et al., 2007] Kim, S., Pakzad, S., Culler, D., Demmel, J., Fenves, G., Glaser, S., and Turon, M. (2007). **Health monitoring of civil infrastructures using wireless sensor networks**. In *Proceedings of the 6th international conference on Information processing in sensor networks*, pages 254–263. ACM.

[Kirby et al., 2011] Kirby, B. T., Ashley-Rollman, M., and Goldstein, S. C. (2011). **Blinky blocks: a physical ensemble programming platform**. In *CHI '11 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '11, pages 1111–1116, New York, NY, USA. ACM.

[Knuth, 1998] Knuth, D. E. (1998). **The art of computer programming: sorting and searching**, volume 3. Pearson Education.

[Kokaji et al., 1996] Kokaji, S., Murata, S., Kurokawa, H., and Tomita, K. (1996). **Clock synchronization mechanisms for a distributed autonomous system**. *J. Robotics and Mechatronics*, 8(5):427–434.

[Korach et al., 1984] Korach, E., Rotem, D., and Santoro, N. (1984). **Distributed algorithms for finding centers and medians in networks**. *ACM Trans. Program. Lang. Syst.*, 6(3):380–401.

[Kusy, 2007] Kusy, B. (2007). **Spatiotemporal coordination in wireless sensor networks**. PhD thesis, Vanderbilt University, Nashville, TN, USA.

[Lakhlef et al., 2015a] Lakhlef, H., and Bourgeois, J. (2015a). **Fast and robust self-organization for micro-electro-mechanical robotic systems**. *Computer Networks*, 93:141–152.

[Lakhlef et al., 2015b] Lakhlef, H., Bourgeois, J., Mabed, H., and Goldstein, S. C. (2015b). **Energy-aware parallel self-reconfiguration for chains microrobot networks**. *Journal of Parallel and Distributed Computing*, 75:67–80.

[Lakhlef et al., 2013] Lakhlef, H., Mabed, H., and Bourgeois, J. (2013). **Distributed and efficient algorithm for self-reconfiguration of MEMS microrobots**. In *SAC 2013, 28th ACM Symposium On Applied Computing*, pages 1–6, Coimbra, Portugal.

[Lakhlef et al., 2014] Lakhlef, H., Mabed, H., and Bourgeois, J. (2014). **Optimization of the logical topology for mobile mems networks**. *Journal of Network and Computer Applications*, 42:163–177.

[Lan et al., 1999] Lan, Y.-F., Wang, Y.-L., and Suzuki, H. (1999). **A linear-time algorithm for solving the center problem on weighted cactus graphs**. *Information Processing Letters*, 71(5-6):205–212.

[Lasagni et al., 2016] Lasagni, M., and Romer, K. (2016). **Dynamic model of tendon-driven robotic chains forming a shape-shifting surface**. In *ASME 2016 Conference on Smart Materials, Adaptive Structures and Intelligent Systems*, page V002T03A024. American Society of Mechanical Engineers.

[Latapy et al., 2006] Latapy, M., and Magnien, C. (2006). **Measuring fundamental properties of real-world complex networks**. *arXiv preprint cs/0609115*.

[Leguay et al., 2005] Leguay, J., Latapy, M., Friedman, T., and Salamatian, K. (2005). **Describing and simulating internet routes**. In *International Conference on Research in Networking*, pages 659–670. Springer.

[Lehmann et al., 2003] Lehmann, K. A., and Kaufmann, M. (2003). **Decentralized algorithms for evaluating centrality in complex networks**.

[Leng et al., 2010] Leng, M., and Wu, Y.-C. (2010). **On clock synchronization algorithms for wireless sensor networks under unknown delay**. *IEEE Transactions on Vehicular Technology*, 59(1):182–190.

[Lenzen et al., 2009] Lenzen, C., Sommer, P., and Wattenhofer, R. (2009). **Optimal clock synchronization in networks**. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 225–238. ACM.

[Lenzen et al., 2015] Lenzen, C., Sommer, P., and Wattenhofer, R. (2015). **Pulsesync: An efficient and scalable clock synchronization protocol**. *IEEE/ACM Transactions on Networking (TON)*, 23(3):717–727.

[Li et al., 2006] Li, Q., and Rus, D. (2006). **Global clock synchronization in sensor networks**. *Computers, IEEE Transactions on*, 55(2):214–226.

[Lynch, 1996] Lynch, N. A. (1996). **Distributed algorithms**. Morgan Kaufmann.

[Ma et al., 2015] Ma, J., Ning, H., Huang, R., Liu, H., Yang, L. T., Chen, J., and Min, G. (2015). **Cybermatics: A holistic field for systematic study of cyber-enabled new worlds**. *IEEE Access*, 3:2270–2280.

[Magnien et al., 2009] Magnien, C., Latapy, M., and Habib, M. (2009). **Fast computation of empirically tight bounds for the diameter of massive graphs**. *Journal of Experimental Algorithmics (JEA)*, 13:10.

[Malpani et al., 2000] Malpani, N., Welch, J. L., and Vaidya, N. (2000). **Leader election algorithms for mobile ad hoc networks**. In *Proceedings of the 4th international workshop on Discrete algorithms and methods for mobile computing and communications*, pages 96–103. ACM.

[Mamei et al., 2005] Mamei, M., Vasirani, M., and Zambonelli, F. (2005). **Self-organizing spatial shapes in mobile particles: The TOTA approach**. In *Engineering Self-Organising Systems*, pages 138–153. Springer.

[Maróti et al., 2004] Maróti, M., Kusy, B., Simon, G., and Lédeczi, Á. (2004). **The flooding time synchronization protocol**. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 39–49. ACM.

[McCarthy, 2000] McCarthy, W. (2000). **Programmable matter**. *Nature*, 407(6804):569–569.

[McEvoy et al., 2015] McEvoy, M. A., and Correll, N. (2015). **Materials that couple sensing, actuation, computation, and communication**. *Science*, 347(6228):1261689.

[Mills, 1991] Mills, D. L. (1991). **Internet time synchronization: the network time protocol**. *Communications, IEEE Transactions on*, 39(10):1482–1493.

[Moffo et al., 2016] Moffo, D., Canalda, P., and Spies, F. (2016). **First evaluation of a system of positioning of microrobots with ultra-dense distribution**. In *7th International conference on Indoor Positioning and indoor Navigation 2016*, Madrid, Espagne.

[Nanda et al., 2008] Nanda, S., and Kotz, D. (2008). **Localized bridging centrality for distributed network analysis**. In *Computer Communications and Networks, 2008. ICCCN'08. Proceedings of 17th International Conference on*, pages 1–6. IEEE.

[Naz et al., 2016] Naz, A., Piranda, B., Goldstein, S. C., and Bourgeois, J. (2016). **Approximate-centroid election in large-scale distributed embedded systems**. In *AINA 2016, 30th IEEE Int. Conf. on Advanced Information Networking and Applications*, pages 548–556, Crans-Montana, Switzerland. IEEE.

[Noh et al., 2007] Noh, K.-L., Chaudhari, Q. M., Serpedin, E., and Suter, B. W. (2007). **Novel clock phase offset and skew estimation using two-way timing message**

exchanges for wireless sensor networks. *IEEE transactions on communications*, 55(4):766–777.

[Oung et al., 2011] Oung, R., and D'Andrea, R. (2011). **The distributed flight array**. *Mechatronics*, 21(6):908–917.

[Park et al., 2008] Park, M., Chitta, S., Teichman, A., and Yim, M. (2008). **Automatic configuration methods in modular robots**. *International Journal for Robotics Research*, 27(3-4):403–421.

[Patterson, 2014] Patterson, S. (2014). **In-network leader selection for acyclic graphs**. *arXiv preprint arXiv:1410.6533*.

[Piranda et al., 2016a] Piranda, B., and Bourgeois, J. (2016a). **A distributed algorithm for reconfiguration of lattice-based modular self-reconfigurable robots**. In *PDP 2016, 24th Euromicro Int. Conf. on Parallel, Distributed, and Network-Based Processing*, pages 1–9, Heraklion Crete, Greece. IEEE.

[Piranda et al., 2016b] Piranda, B., and Bourgeois, J. (2016b). **Geometrical study of a quasi-spherical module for building programmable matter**. In *DARS 2016, 13th Int. Symposium on Distributed Autonomous Robotic Systems*. Springer.

[Piranda et al., 2013] Piranda, B., Laurent, G. J., Bourgeois, J., Clévy, C., Möbes, S., and Le Fort-Piat, N. (2013). **A new concept of planar self-reconfigurable modular robot for conveying microparts**. *Mechatronics*, 23(7):906–915.

[Raynal, 2013] Raynal, M. (2013). **Distributed algorithms for message-passing systems**, volume 500. Springer.

[Reynolds et al., 1994] Reynolds, J. K., and Postel, J. (1994). **Assigned Numbers**. RFC 1700, RFC Editor.

[Roditty et al., 2013] Roditty, L., and Vassilevska Williams, V. (2013). **Fast approximation algorithms for the diameter and radius of sparse graphs**. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 515–524. ACM.

[Römer et al., 2005] Römer, K., Blum, P., and Meier, L. (2005). **Time synchronization and calibration in wireless sensor networks**. *Handbook of sensor networks: Algorithms and architectures*, 49:199.

[Rubenstein et al., 2014] Rubenstein, M., Cornejo, A., and Nagpal, R. (2014). **Programmable self-assembly in a thousand-robot swarm**. *Science*, 345(6198):795–799.

[Şahin, 2004] Şahin, E. (2004). **Swarm robotics: From sources of inspiration to domains of application**. In *International workshop on swarm robotics*, pages 10–20. Springer.

[Schenato et al., 2011] Schenato, L., and Fiorentin, F. (2011). **Average timesynch: A consensus-based protocol for clock synchronization in wireless sensor networks**. *Automatica*, 47(9):1878–1886.

[Shimbel, 1953] Shimbel, A. (1953). **Structural parameters of communication networks**. *The bulletin of mathematical biophysics*, 15(4):501–507.

[Sommer et al., 2009] Sommer, P., and Wattenhofer, R. (2009). **Gradient clock synchronization in wireless sensor networks**. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, pages 37–48. IEEE Computer Society.

[Stoy, 2003] Stoy, K. (2003). **Emergent Control of Self-Reconfigurable Robots**. PhD thesis, University of Southern Denmark.

[Stoy et al., 2011] Stoy, K., and Kurokawa, H. (2011). **Current topics in classic self-reconfigurable robot research**. In *Proceedings of the IROS Workshop on Reconfigurable Modular Robotics: Challenges of Mechatronic and Bio-Chemo-Hybrid Systems*.

[Stoy et al., 2002a] Stoy, K., Shen, W.-M., and Will, P. (2002a). **Global locomotion from local interaction in self-reconfigurable robots**. In *Proc. of the 7th Intl. Conf. on Intelligent Autonomous Systems (IAS-7)*, pages 309–316.

[Stoy et al., 2002b] Stoy, K., Shen, W.-M., and Will, P. (2002b). **How to make a self-reconfigurable robot run**. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*, pages 813–820. ACM.

[Su et al., 2005] Su, W., and Akyildiz, I. F. (2005). **Time-diffusion synchronization protocol for wireless sensor networks**. *Networking, IEEE/ACM Transactions on*, 13(2):384–397.

[Suh et al., 2002] Suh, J. W., Homans, S. B., and Yim, M. (2002). **Telecubes: Mechanical design of a module for self-reconfigurable robotics**. In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, volume 4, pages 4095–4101. IEEE.

[Syed et al., 2006] Syed, A. A., and Heidemann, John S, e. a. (2006). **Time synchronization for high latency acoustic networks.** In *Infocom*, volume 6, pages 1–12.

[Takes et al., 2013] Takes, F. W., and Kosters, W. A. (2013). **Computing the eccentricity distribution of large graphs**. *Algorithms*, 6(1):100–118.

[The Internet Assigned Numbers Authority (IANA), 2016] The Internet Assigned Numbers Authority (IANA) (2016). **Internet protocol version 4 (IPv4) parameters**. http://www.iana.org/assignments/ip-parameters/ip-parameters.xhtml.

[Tibbits et al., 2014] Tibbits, S., McKnelly, C., Olguin, C., Dikovsky, D., and Hirsch, S. (2014). **4d printing and universal transformation**. In *34th Annual Conference of the Association for Computer Aided Design in Architecture (ACADIA)*, pages 539–548, Los Angeles, CA, USA.

[Tizghadam et al., 2010] Tizghadam, A., and Leon-Garcia, A. (2010). **Betweenness centrality and resistance distance in communication networks**. *Network, IEEE*, 24(6):10–16.

[Toueg, 1980] Toueg, S. (1980). **An all-pairs shortest paths distributed algorithm**. Technical Report, IBM.

[Tucci et al., 2017] Tucci, T., Piranda, B., and Bourgeois, J. (2017). **Efficient scene encoding for programmable matter self-reconfiguration algorithms**. In *Proceedings of the Symposium on Applied Computing*, pages 256–261. ACM.

[Varagnolo et al., 2010] Varagnolo, D., Pillonetto, G., and Schenato, L. (2010). **Distributed statistical estimation of the number of nodes in sensor networks**. In *Decision and Control (CDC), 2010 49th IEEE Conference on*, pages 1498–1503. IEEE.

[Vasudevan et al., 2004] Vasudevan, S., Kurose, J., and Towsley, D. (2004). **Design and analysis of a leader election algorithm for mobile ad hoc networks**. In *Network Protocols, 2004. ICNP 2004. Proceedings of the 12th IEEE International Conference on*, pages 350–360. IEEE.

[Vermesan et al., 2017] Vermesan, O., Bröring, A., Tragos, E., Serrano, M., Bacciu, D., Chessa, S., Gallicchio, C., Micheli, A., Dragone, M., Saffiotti, A., and others (2017). **Internet of robotic things: converging sensing/actuating, hypoconnectivity, artificial intelligence and iot platforms**. In *Cognitive hyperconnected digital transformation: internet of things intelligence evolution*, pages 1–35. River Publishers.

[Vermesan et al., 2013] Vermesan, O., and Friess, P. (2013). **Internet of things: converging technologies for smart environments and integrated ecosystems**. River Publishers.

[Walter et al., 2005] Walter, J. E., Tsai, E. M., and Amato, N. M. (2005). **Algorithms for fast concurrent reconfiguration of hexagonal metamorphic robots**. *IEEE transactions on Robotics*, 21(4):621–631.

[Walter et al., 2000] Walter, J. E., Welch, J. L., and Amato, N. M. (2000). **Distributed reconfiguration of metamorphic robot chains**. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 171–180. ACM.

[Wang et al., 2015] Wang, W., and Tang, C. Y. (2015). **Distributed estimation of closeness centrality**. In *Decision and Control (CDC), 2015 IEEE 54th Annual Conference on*, pages 4860–4865. IEEE.

[Wasserman, 1994] Wasserman, S. (1994). **Social network analysis: Methods and applications**, volume 8. Cambridge university press.

[Watts et al., 1998] Watts, D. J., and Strogatz, S. H. (1998). **Collective dynamics of 'small-world'networks**. *Nature*, 393(6684):440–442.

[Wehmuth et al., 2011] Wehmuth, K., and Ziviani, A. (2011). **Distributed assessment of network centrality**. *CoRR, abs/1108.1067*.

[Wehmuth et al., 2013] Wehmuth, K., and Ziviani, A. (2013). **Daccer: Distributed assessment of the closeness centrality ranking in complex networks**. *Computer Networks*, 57(13):2536–2548.

[Wong et al., 2013] Wong, S., and Walter, J. (2013). **Deterministic distributed algorithm for self-reconfiguration of modular robots from arbitrary to straight chain configurations**. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 537–543. IEEE.

[Wong et al., 2015] Wong, S., Zhu, S., and Walter, J. (2015). **Unpacking a cluster of modular robots**. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 103. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp).

[Wuchty et al., 2003] Wuchty, S., and Stadler, P. F. (2003). **Centers of complex networks**. *Journal of Theoretical Biology*, 223(1):45–53.

[Yim et al., 2007] Yim, M., Shen, W.-M., Salemi, B., Rus, D., Moll, M., Lipson, H., Klavins, E., and Chirikjian, G. S. (2007). **Modular self-reconfigurable robot systems [grand challenges of robotics]**. *Robotics & Automation Magazine, IEEE*, 14(1):43–52.

[Yim et al., 2009] Yim, M., White, P., Park, M., and Sastra, J. (2009). **Modular self-reconfigurable robots**. In *Encyclopedia of complexity and systems science*, pages 5618–5631. Springer.

[You et al., 2017] You, K., Tempo, R., and Qiu, L. (2017). **Distributed algorithms for computation of centrality measures in complex networks**. *IEEE Transactions on Automatic Control*, 62(5):2080–2094.

[Zhao et al., 2007] Zhao, Y., Chen, Y., Li, B., and Zhang, Q. (2007). **Hop id: A virtual coordinate based routing for sparse mobile ad hoc networks**. *Mobile Computing, IEEE Transactions on*, 6(9):1075–1089.

APPENDICES

A

Demonstrations of LMR Network Properties

Contents

A.1/ INTRODUCTION

In this appendix, we demonstrate that LMRs form sparse and large-diameter networks. Moreover, we provide exact bounds on the radius and the diameter of these networks based on their lattice type and the number of modules in the system.

We illustrate our demonstrations using the modular robots designed in the Smart Blocks and the Claytronics projects, namely the Smart Blocks, the millimeter-scale 2D Catoms, the Blinky Blocks and the 3D Catoms [Piranda et al., 2016b] (see Figure 2.3). These modular robots are arranged in the square, the hexagonal, the simple cubic, and the face-centered cubic lattices, respectively.

The analysis of the 3D-Catom system radius presented in this appendix was realized in cooperation with Thadeu Tucci, my office mate and PhD student.

The rest of this appendix is organized as follows. Section A.2 presents the related work. Then, section A.3 defines the system model and some terms. Afterwards, Section A.4 characterizes the network density for our class of modular robots. Section A.5 provides tight bounds of the radius and the diameter of the networks for our class of modular robots.

A.2/ RELATED WORK

To the best of our knowledge, little attraction has been paid to network characterization in the modular robotic community. In [Garcia et al., 2009], the authors compare the efficiency of neighbor-to-neighbor communication and global communication. Based on experimentally validated models, the authors compare the information transmission time in different scenarios for systems composed of 10 to 1000 modules. As mentioned in Section 2.2.4, global communication through a shared medium is less scalable with system size. Since we envision systems composed of millions of units, global communication is not an option.

As characterizing network properties is crucial for choosing appropriate algorithms and designing efficient new ones, graphs and networks have been extensively studied. Studies have been conducted on various graphs and networks, e.g., the Internet [Latapy et al., 2006, Cardozo et al., 2012, Jin et al., 2006], the World Wide Web [Albert et al., 1999], sensor networks [Jennings et al., 2002], small-world networks [Watts et al., 1998, Hayes, 2000], unit disk graphs [Ellis et al., 2004], and lattice-based networks [Hayes, 2000, Barrenetxea et al., 2006, Barthélemy, 2011]. These studies are network-specific. They are either measurement-based (e.g., [Latapy et al., 2006, Cardozo et al., 2012, Albert et al., 1999]), or purely theoretical using the intrinsic characteristics of the network (e.g., [Jennings et al., 2002, Ellis et al., 2004, Barrenetxea et al., 2006, Barthélemy, 2011]).

Due to the regular tiling of the space in lattices, lattice-based networks obey certain geometric rules that can be used to analyze these networks. In [Hayes, 2000, Barrenetxea et al., 2006], the authors study some lattice-based networks, but they only consider net-

works embedded in the square lattice and restrict their analysis to specific network topologies, e.g., the square, the ring, etc. Their results are not generalizable to other lattices and arbitrary network topologies. In [Barthélemy, 2011], the author states that the average distance between nodes in lattice networks is on the order of $n^{\frac{1}{D_L}}$, where n is the number of nodes and D_L is the dimension of the considered lattice.

In this appendix, we consider lattice-based networks embedded in any of the square, hexagonal, simple-cubic and face-centered lattices. We show that these networks are sparse and have a large diameter. Moreover, we provide tight lower and upper bounds for the radius and the diameter of these networks.

A.3/ System Model and Definitions

In LMRs, modules are arranged in some regular 2-dimensional or 3-dimensional lattice L. Here, we consider the Square (S), the Hexagonal (H), the Simple Cubic (SC) and the Face-Centered Cubic (FCC) lattices. Modules can only occupy a set of discrete positions defined by L. Note that modular robots may contain holes, i.e., some positions of L may be unoccupied. As we assume neighbor-to-neighbor communications, L also defines the module connectivity: Modules can directly communicate only with their immediate neighbors in L. D_L denotes the dimension of L and Δ_L represents its coordination number, i.e, the maximum number of modules to which a module can be connected.

Arbitrarily arranged modular robotic systems form lattice-based networks that can be modeled by connected, undirected, unweighted and lattice-based graphs $G = (V, E)$, where V is the set of vertices (representing the modules), E the set of edges (representing the connections), $|V| = n$, the number of vertices and $|E| = m$, the number of edges. $\delta(v_i)$ denotes v_i's degree, i.e., the number of vertices to which v_i is connected. $d(v_i, v_j)$ refers to the distance between the vertices v_i and v_j, i.e., the number of edges on a shortest path between v_i and v_j. The radius, r, and the diameter, d, of G are respectively defined as $r = \min_{v_i \in V} \max_{v_j \in V} d(v_i, v_j)$ and $d = \max_{v_i \in V} \max_{v_j \in V} d(v_i, v_j)$.

Notice that we assume a perfect alignment of the modules in the lattice. However, defects in the lattice, which may cause unreliable and intermittent connections, will only make the network sparser and increase both its radius and its diameter.

We now define some specific graphs used in this chapter. Let V_L be the infinite set of vertices representing the infinite set of positions in L. The $L\text{-}Sphere(v_c, r)$ is a sphere embedded in L, where the vertex v_c is the center of the sphere and $r \in \mathbb{N}$ its radius. It contains the set of vertices in V_L whose distance from v_c is equal to r:

$$L\text{-}Sphere(v_c, r) = \{v_i \in V_L \mid d(v_i, v_c) = r\} \tag{A.1}$$

$L\text{-}Ball(v_c, r)$ is a ball embedded in L, where v_c is the center of the ball and $r \in \mathbb{N}$ is its radius. It contains the set of vertices in V_L whose distance from v_c is less than or equal

to r:

$$L\text{-}Ball(v_c, r) = \{v_i \in V_L \mid d(v_i, v_c) \leq r\} \tag{A.2}$$

$$= \bigcup_{i=0}^{r} L\text{-}Sphere(v_c, i) \tag{A.3}$$

By an abuse of notation, $L\text{-}Sphere$ and $L\text{-}Ball$ can respectively refer to sphere and ball graphs embedded in L where the connectivity between vertices is induced by the lattice structure of L. $L\text{-}Sphere(r)$ and $L\text{-}Ball(r)$ respectively refer to a sphere and a ball of radius r in the lattice L. In all the illustrations of this chapter, $L\text{-}Sphere(r)$ is gradually colored from red to blue according to the value of r.

A.4/ Network Density

In this section, we show that the networks formed by our class of modular robots are all sparse.

Corollary A.4.1: Let $G = (V, E)$ be the network graph of an arbitrarily arranged modular robotic system that fits the model described in section A.3. The vertex degree, $\delta(v_i)$, of any vertex $v_i \in V$ is bounded by:

$$0 \leq \delta(v_i) \leq \Delta_L \tag{A.4}$$

Lemma A.4.1: Let $G = (V, E)$ be the network graph of an arbitrarily arranged modular robotic system that fits the model described in section A.3. The number of edges of G, m, is bounded as follows:

$$n - 1 \leq m \leq n\Delta_L \tag{A.5}$$

Proof. **Lower Bound.** A connected graph must have at least n-1 edges [Hayes, 2000].

Upper Bound. Because of Corollary A.4.1, every module cannot be connected to more than Δ_L others. Thus, the number of edges of G is upper-bounded by $n\Delta_L$. Note that a tighter upper bound can be established by considering the lattice structure of L.

Theorem A.4.1: Let $G = (V, E)$ be the network graph of an arbitrarily arranged modular robotic system that fits the model described in section A.3. If $|V| = n$ is large, then G is a sparse graph, i.e., $m \ll n^2$.

Proof. If n is large, then $\Delta_L \ll n$. Thus, we have $n\Delta_L \ll n^2$. Then, because of Lemma A.4.1, we obtain $m \ll n^2$.

A.5/ Network Radius and Diameter

In this section, we establish tight lower and upper bounds of the radius and the diameter of the networks of our class of modular robots.

A.5.1/ PRELIMINARY MATERIALS

This section presents some preliminary results used in the computations and the demonstrations of the radius and the diameter bounds of modular robot networks. We recall that V_L is the infinite set of vertices representing the set of positions in the lattice L.

Corollary A.5.1: $\forall v_c \in V_L,\ \forall r \in \mathbb{N},\ L\text{-}Ball(v_c, r)$ is centrally symmetric: The reflection v_j of every vertex v_i at distance $d(v_i, v_c) = k$ through v_c is also at distance k from v_c and $d(v_i, v_j) = 2k$.

Proof. Let $L\text{-}Ball(v_c, 1)$ be the ball of radius 1 and v_c its center. All the vertices except v_c are at distance 1 from v_c. Along every axis of the lattice L, two vertices, v_1 and v_2 are connected to v_c, one in each direction. These two vertices are symmetric through v_c, at distance 1 from v_c and at distance 2 from each other.

Let $L\text{-}Ball(v_c, r)$ be the ball of radius r and v_c its center. We assume that $L\text{-}Ball(v_c, r)$ is centrally symmetric. Let $L\text{-}Ball(v_c, r+1)$ be the ball of radius $r+1$ with v_c being its center. By construction, $L\text{-}Ball(v_c, r+1)$ is obtained from $L\text{-}Ball(v_c, r)$ by adding all the vertices at distance $r+1$ from v_c. Let us consider v_3 and v_4 in $L\text{-}Ball(v_c, r)$ such that v_3 and v_4 are symmetric through v_c and $d(v_3, v_4) = 2r$. In order to construct $L\text{-}Ball(v_c, r+1)$, we add to v_3 and v_4 two vertices v_5 and v_6 on the same axis but in the opposite direction such that $d(v_5, v_c) = d(v_6, v_c) = r+1$. v_5 and v_6 are symmetric through v_c. Moreover, there is no shortcut between v_5 and v_6, thus, $d(v_5, v_6) = 1 + d(v_3, v_4) + 1 = 2 + 2r = 2(r+1)$. Thus, $L\text{-}Ball(v_c, r+1)$ is centrally symmetric.

By induction, $\forall v_c \in V_L,\ \forall r \in \mathbb{N},\ L\text{-}Ball(v_c, r)$ is centrally symmetric.

Lemma A.5.1: $\forall v_c \in V_L,\ \forall r \in \mathbb{N}$, the diameter, d, of $L\text{-}Ball(v_c, r)$ is equal to $2r$.

Proof. As stated in Corollary A.5.1, $L\text{-}Ball(v_c, r)$ is centrally symmetric. Thus, $\forall v_i \in L\text{-}Ball(v_c, r)$ such that $d(v_i, v_c) = r$, $\exists v_j \in L\text{-}Ball(v_c, r)$ with $d(v_i, v_j) = 2r$. By construction, $\nexists v_i \in L\text{-}Ball(v_c, r),\ d(v_i, v_c) > r$. As a consequence, the diameter of $L\text{-}Ball(v_c, r)$, i.e., the largest distance between any two vertices, is equal to $d = 2r$.

Corollary A.5.2: $\forall v_c \in V_L,\ \forall r \in \mathbb{N},\ L\text{-}Ball(v_c, r)$ is the minimum-radius and minimum-diameter existing graph composed of $n_{L\text{-}Ball}(v_c, r) = |\,L\text{-}Ball(v_c, r)|$ vertices in L.

Proof. By construction, in $L\text{-}Ball(v_c, r)$ all the positions of the lattice L at a distance less than or equal to r from v_c are occupied. Thus, if we remove a vertex v_1 and add it to an empty place adjacent to a full one (the system should remain connected) occupied by the vertex v_2, the new location of v_1 must be at distance $r+1$ from v_c. Moreover, every vertex would be at distance $r+1$ or more from at least one other vertex. Thus, the radius of the graph would be equal to $r+1$. Moreover, because $L\text{-}Ball(v_c, r)$ is centrally symmetric (See Corollary A.5.1), $\exists v_3 \in L\text{-}Ball(v_c, r),\ d(v_2, v_3) = 2r$. Because of Lemma A.5.1, $d(v_2, v_3)$ is the diameter of $L\text{-}Ball(v_c, r)$. Since there is no shortcut between v_1 and v_3 in its new location, $d(v_1, v_3) = d(v_2, v_3) + 1 = 2r + 1$. Thus, the diameter of the graph would be equal to $2r+1$.

A.5.2/ Radius and Diameter Bounds

Theorem A.5.1: Let $G = (V, E)$ be the network graph of an arbitrarily arranged modular robotic system that fits the model described in section A.3. Let $L\text{-}Ball(r_b)$ and $L\text{-}Ball(r_b+1)$ be two ball graphs embedded in L, such that the number of vertices of G, n, is between the number of vertices of these two balls, i.e., $n_{L\text{-}Ball}(r_b) \leq n < n_{L\text{-}Ball}(r_b + 1)$. The radius, r, and the diameter, d, of G are tightly bounded as follows:

$$r_b \leq r \leq \lfloor \frac{n-1}{2} \rfloor \tag{A.6}$$

$$2r_b \leq d \leq n - 1 \tag{A.7}$$

Proof. **Upper Bound.** In a connected graph, any two vertices are at most separated by all the others. In such a graph, the n vertices form a line of $n-1$ edges. Thus, the largest distance between any two vertices, i.e., the diameter of G, is at most equal to $n-1$ edges. The radius of G is at most equal to the half of that line, i.e, $r \leq \lfloor \frac{n-1}{2} \rfloor$.
Lower Bound. Because of Corollary A.5.2, $L\text{-}Ball(r_b)$ is the minimum-radius and minimum-diameter graph composed of $n_{L\text{-}Ball}(r_b)$ vertices. Thus, with n vertices, G has a radius at least equal to r_b and a diameter at least equal to the diameter of $L\text{-}Ball(r_b)$, which is, because of Lemma A.5.1, equal to $2r_b$.

In the rest of this section, we establish the formula to compute the exact radius of an $L\text{-}Ball$ according to its number of vertices in the different lattices considered.

Systems in Two Dimensions: The Square and Hexagonal Lattices In this section, we compute the exact radius of an $L\text{-}Ball$, given the number of vertices it has, for the case of two-dimensional systems embedded in the Square (S) and Hexagonal (H) lattices. Figure A.1 depicts an $S\text{-}Ball$ and an $H\text{-}Ball$ of radius 4, composed of Smart Blocks and 2D Catoms, respectively.

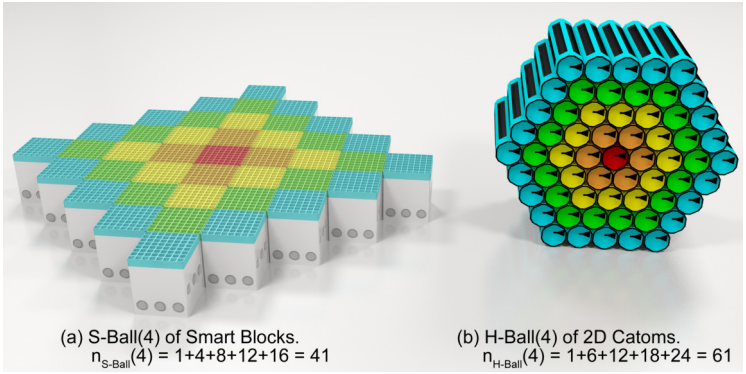


Figure A.1: An $S\text{-}Ball(4)$ and an $H\text{-}Ball(4)$ with color gradient from the center of the ball.

Lemma A.5.2: In the square and the hexagonal lattices, the number of vertices in a

sphere of radius $r \geq 1$, $n_{L\text{-}Sphere}(r, \Delta_L)$, can be computed by:

$$n_{L\text{-}Sphere}(r, \Delta_L) = r\Delta_L \tag{A.8}$$

Proof. As illustrated in Figure A.1, in the square and the hexagonal lattices, a sphere of radius $r \geq 1$ is composed of Δ_L segments of length r modules. Consequently, the number of vertices is equal to $r\Delta_L$.

Theorem A.5.2: In the square and the hexagonal lattices, the radius of a ball composed of $n \geq 1$ vertices, $r_{L\text{-}Ball}(n, \Delta_L)$, can be computed by:

$$r_{L\text{-}Ball}(n, \Delta_L) = \frac{1}{2}\left(\sqrt{1 + \frac{8(n-1)}{\Delta_L}} - 1\right) \tag{A.9}$$

Proof. By definition, $L\text{-}Ball(r)$ is the union of all the $L\text{-}Sphere(i)$ for i ranging from 0 to r. Thus, in the square and the hexagonal lattices, for $r \geq 1$, the number of vertices in an $L\text{-}Ball(r)$, $n_{L\text{-}Ball}(r, \Delta_L)$, can be computed as follows:

$$n_{L\text{-}Ball}(r, \Delta_L) = \sum_{i=0}^{r} n_{L\text{-}Sphere}(i, \Delta_L) \tag{A.10}$$

$$= 1 + \sum_{i=1}^{r} i\Delta_L \tag{A.11}$$

$$= \frac{1}{2}r^2\Delta_L + \frac{1}{2}r\Delta_L + 1 \tag{A.12}$$

To obtain Equation A.9, we solve Equation (A.12) for r and keep only the positive root.

Systems in Three Dimensions: The Simple Cubic and Face-Centered Cubic Lattices In this section, we compute the exact radius of an $L\text{-}Ball$, given the number of vertices it contains, for the case of three-dimensional systems embedded in the Simple Cubic (SC) and Face-Centered Cubic (FCC) lattices. Figures A.2 and A.3 depict the $SC\text{-}Ball$ and the $FCC\text{-}Ball$ of radius 2, composed of Blinky Blocks and 3D Catoms, respectively. Both systems can be decomposed into horizontal layers.

The Simple Cubic Lattice

Lemma A.5.3: In the simple cubic lattice, the number of vertices in a sphere of radius $r \geq 1$, $n_{SC\text{-}Sphere}(r)$, can be computed by:

$$n_{SC\text{-}Sphere}(r) = n_{S\text{-}Sphere}(r) + 2\sum_{i=0}^{r-1} n_{S\text{-}Sphere}(i) \tag{A.13}$$

$$= 2(2r^2 + 1) \tag{A.14}$$

Proof. As illustrated in Figure A.2, a sphere of radius r in the simple cubic lattice can be decomposed into $2r+1$ horizontal $S\text{-}Sphere\ s$ of different radii. Equation (A.13) is obtained by summing up all the sizes of the $S\text{-}Sphere\ s$.

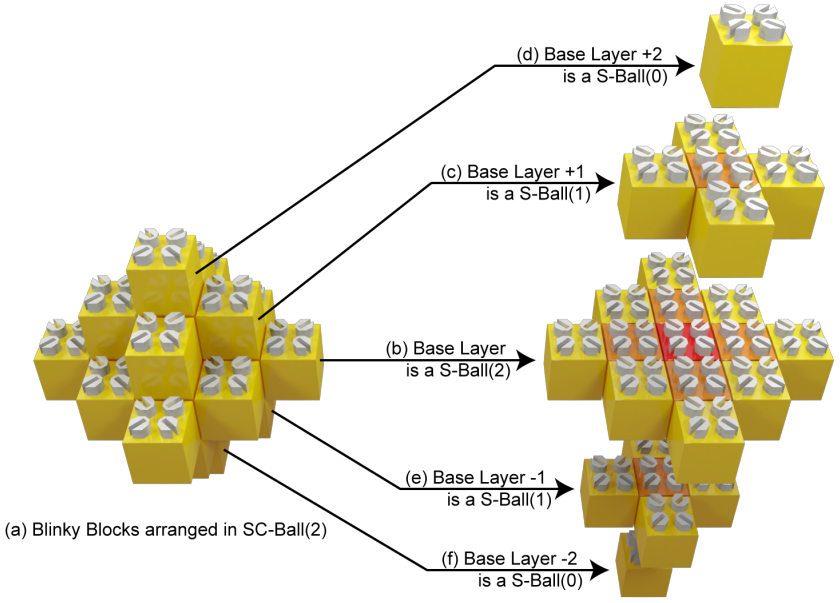


Figure A.2: An $SC\text{-}Ball(2)$ of Blinky Blocks and its decomposition into horizontal layers with color gradient from the center of the ball.

Theorem A.5.3: In the simple-cubic lattice, the radius of a ball composed of $n \geq 1$ vertices, $r_{SC\text{-}Ball}(n)$, can be computed by:

$$r_{SC\text{-}Ball}(n) = \frac{1}{2}\left(\frac{(\sqrt{3}\sqrt{243n^2+125}+27n)^{\frac{1}{3}}}{3^{\frac{2}{3}}} - \frac{5}{3^{\frac{1}{3}}(\sqrt{3}\sqrt{243n^2+125}+27n)^{\frac{1}{3}}} - 1\right) \tag{A.15}$$

Proof. By definition, $L\text{-}Ball(r)$ is the union of all the $L\text{-}Sphere(i)$ for i ranging from 0 to r. Thus, for $r \geq 1$, the number of vertices in an $SC\text{-}Ball(r)$, $n_{SC\text{-}Ball}(r)$, can be computed as follows:

$$n_{SC\text{-}Ball}(r) = \sum_{i=0}^{r} n_{SC\text{-}Sphere}(i) \tag{A.16}$$

$$= 1 + \sum_{i=1}^{r} 2(2i^2+1) \tag{A.17}$$

$$= \frac{4}{3}r^3 + 2r^2 + \frac{8}{3}r + 1 \tag{A.18}$$

To obtain Equation A.15, we solve Equation (A.18) for r and keep only the real root.

The Face-Centered Cubic Lattice

Lemma A.5.4: In the face-centered cubic lattice, the number of vertices in a sphere of radius $r \geq 1$, $n_{FCC\text{-}Sphere}(r)$, can be computed by:

$$n_{FCC\text{-}Sphere}(r) = 4r + 2(r+1)^2 + 2(r-1)4r \tag{A.19}$$

$$= 2(5r^2+1) \tag{A.20}$$

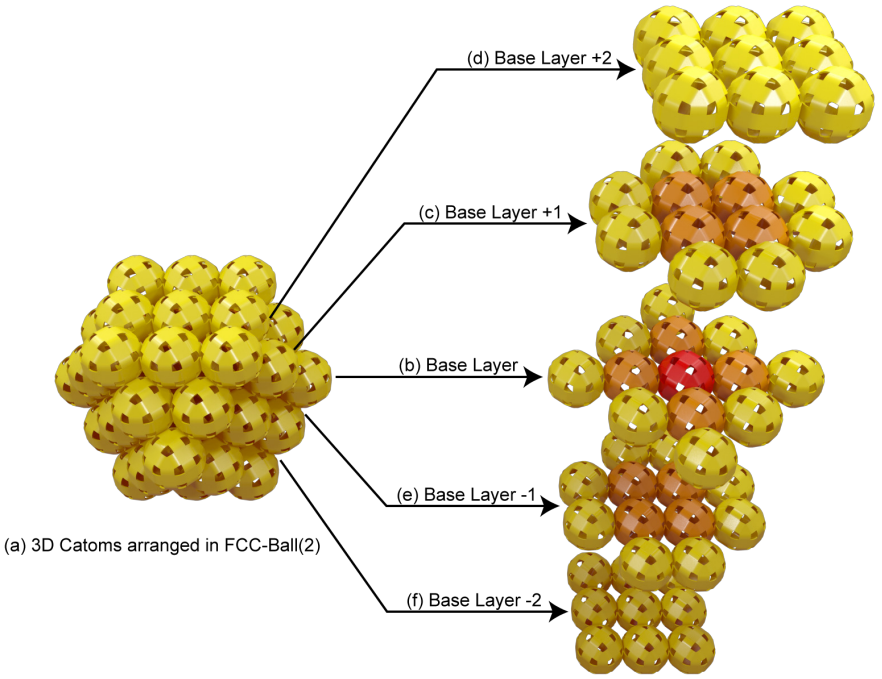


Figure A.3: An $FCC\text{-}Ball(2)$ of 3D Catoms and its decomposition into horizontal layers with color gradient from the center of the ball.

Proof. As shown in Figure A.3, a sphere of radius r in the face-centered cubic lattice can be decomposed into $2r+1$ horizontal layers. The base layer is an $S\text{-}Sphere(r)$ and contains $4r$ vertices. The bottom and the top layers both contain $(r+1)^2$ vertices. The $2(r-1)$ other layers contain $4r$ vertices each. Equation (A.19) is obtained by summing up the number or vertices of each layer.

Theorem A.5.4: In the face-centered cubic lattice, the radius of a ball of $n \geq 1$ vertices, $r_{FCC\text{-}Ball}(n)$, can be computed by:

$$r_{FCC\text{-}Ball}(n) = \frac{1}{2}\left(\frac{(\sqrt{15}\sqrt{4860n^2+343}+270n)^{\frac{1}{3}}}{15^{\frac{2}{3}}} - \frac{7}{15^{\frac{1}{3}}(\sqrt{15}\sqrt{4860n^2+343}+270n)^{\frac{1}{3}}} - 1\right) \tag{A.21}$$

Proof. By definition, $L\text{-}Ball(r)$ is the union of all the $L\text{-}Sphere(i)$ for i ranging from 0 to r. Thus, for $r \geq 1$, the number of vertices in an $FCC\text{-}Ball(r)$, $n_{FCC\text{-}Ball}(r)$, can be computed as follows:

$$n_{FCC\text{-}Ball}(r) = \sum_{i=0}^{r} n_{FCC\text{-}Sphere}(i) \tag{A.22}$$

$$= 1 + \sum_{i=1}^{r} 2(5i^2+1) \tag{A.23}$$

$$= \frac{10}{3}r^3 + 5r^2 + \frac{11}{3}r + 1 \tag{A.24}$$

To obtain Equation A.21, we solve Equation (A.24) for r and keep only the real root.

Document generated with LaTeX and:
the LaTeX style for PhD Thesis created by S. Galland — http://www.multiagent.fr/ThesisStyle
the tex-upmethodology package suite — http://www.arakhne.org/tex-upmethodology/

école doctorale sciences pour l'ingénieur et microtechniques

Université Bourgogne Franche-Comté
32, avenue de l'Observatoire
25000 Besanon, France

Title: Distributed Algorithms for Large-Scale Robotic Ensembles: Centrality, Synchronization and Self-Reconfiguration

Keywords: Distributed algorithms, Modular robotics, Centrality-based leader election, Time synchronization, Self-reconfiguration.

Abstract:

Technological advances, especially in the miniaturization of robotic devices foreshadow the emergence of large-scale ensembles of small-size resource-constrained robots that distributively cooperate to achieve complex tasks (e.g., modular self-reconfigurable robots, swarm robotic systems, distributed microelectromechanical systems, etc.). These ensembles are formed by independent, intelligent and communicating units which act as a whole ensemble. These units cooperatively self-organize themselves to achieve common goals. These systems are thought to be more versatile and more robust than conventional robotic systems while having at the same time a lower cost. These ensembles form complex asynchronous distributed systems in which every unit is an embedded system with its own but limited capabilities. Coordination of such large-scale distributed embedded systems poses significant algorithmic issues and open new opportunities in distributed algorithms. In my thesis, I defend the idea that distributed algorithmic primitives suitable for the coordination of these ensembles should be both identified and designed. In this work, we focus on a specific class of modular robotic systems, namely large-scale distributed modular robotic ensembles composed of resource-constrained modules that are organized in a lattice structure and which can only communicate with neighboring modules. We identified and implemented three building blocks, namely centrality-based leader election, time synchronization and self-reconfiguration. We propose a collection of distributed algorithms to realize these primitives. We evaluate them using both hardware experiments and simulations on systems ranging from a dozen modules to more than ten thousand modules. We show that our algorithms scale well and are suitable for large-scale embedded distributed systems with scarce memory and computing resources.

Titre : Algorithmes distribués pour grands ensembles de robots : centralité, synchronization et auto-reconfiguration

Mots-clés : algorithmique distribuée, robots modulaires, élection de leader basée sur la centralité, synchronisation temporelle, auto-reconfiguration.

Résumé :

Les récentes avancées technologiques, en particulier dans le domaine de la miniaturisation de dispositifs robotiques, laissent présager l'émergence de grands ensembles distribués de petits robots qui coopéreront en vue d'accomplir des tâches complexes (e.g., robotique modulaire, robots en essaims, microsystèmes électromécaniques distribués). Ces grands ensembles seront composés d'entités indépendantes, intelligentes et communicantes qui agiront comme un ensemble à part entière. Pour cela, elles s'auto-organiseront et collaboreront en vue d'accomplir des tâches complexes. Ces systèmes présenteront les avantages d'être plus polyvalents et plus robustes que les systèmes robotiques conventionnels tout en affichant un prix réduit. Ces ensembles formeront des systèmes distribués complexes dans lesquels chaque entité sera un système embarqué à part entière avec ses propres capacités et ressources toutefois limitées. Coordonner de tels systèmes pose des défis majeurs et ouvre de nouvelles opportunités dans l'algorithmique distribuée. Je défends la thèse qu'il faut d'ores et déjà identifier et implémenter des algorithmes distribués servant de primitives de base à la coordination de ces ensembles. Dans ce travail, nous nous focalisons sur une classe particulière de robots, à savoir les robots modulaires distribués formant de grands ensembles de modules fortement contraints en ressources (mémoire, calculs, etc.), placés dans une grille régulière et capables de communiquer entre voisins connexes uniquement. J'ai identifié et implémenté trois primitives servant à la coordination de ces systèmes, à savoir l'élection d'un nœud central au réseau, la synchronisation temporelle ainsi que l'auto-reconfiguration. Dans ce manuscrit, je propose un ensemble d'algorithmes distribués réalisant ces primitives. Les algorithmes développés dans le cadre de ce travail ont été évalués sur des modules matériels et par simulation avec des systèmes composés de quelques dizaines à plus d'une dizaine de milliers de modules. Ces expériences montrent que nos algorithmes passent à l'échelle et sont adaptés aux grands ensembles distribués de systèmes embarqués avec des ressources fortement limitées à la fois en mémoire et en calcul.