Efficient Distributed Algorithms Suited for Uncertain Contexts

Anaïs Durand

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 25 Mai 2016

Présentée par

Anaïs DURAND

Thèse dirigée par **Karine ALTISEN**
et codirigée par **Stéphane DEVISMES**

préparée au sein de **Verimag**
et de l'**École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Efficient Distributed Algorithms Suited for Uncertain Contexts

Algorithmes distribués efficaces adaptés à un contexte incertain

Thèse soutenue publiquement le **1er Septembre 2017**,
devant le jury composé de :

Pierre FRAIGNIAUD
Directeur de Recherche, CNRS, Président
Christian SCHEIDELER
Professeur, Universität Paderborn, Rapporteur
Sébastien TIXEUIL
Professeur, Université Pierre et Marie Curie - Paris 6, Rapporteur
Paola FLOCCHINI
Professeur, Université d'Ottawa, Examinatrice
Colette JOHNEN
Professeur, Université de Bordeaux, Examinatrice
Michel RAYNAL
Professeur, Université de Rennes 1, Examinateur
Karine ALTISEN
Maître de Conférences, Grenoble INP, Directrice de thèse
Stéphane DEVISMES
Maître de Conférences, Université Grenoble Alpes, Co-Directeur de thèse

Remerciements

"Isn't it odd how the little things can change a man's entire life ?"
— David Eddings, *Belgarath the Sorcerer*

Si ma thèse n'a duré que trois ans, c'est une aventure de huit longues années qui s'achève aujourd'hui. Cette aventure commence pendant le rendez-vous pédagogique pour ma première inscription à l'université de Grenoble, lorsque l'un des enseignants me dit : "Avec votre profil, vous devriez essayer le pré-Magistère. Ça devrait vous plaire." Puis lorsqu'un stage d'excellence me mène à me perdre au fond du campus, sous une pluie battante, à la recherche d'un laboratoire, Verimag, où j'allais passer de nombreux mois au gré de stages et pendant ma thèse. Ce n'est donc pas sans émotions que je vais bientôt quitter les couloirs de Verimag et les montagnes de Grenoble.

Je tiens tout d'abord à remercier **Karine** et **Stéphane** qui m'ont fait découvrir l'algorithmique distribuée et la recherche. Tout au long de ces huit années, pendant mes stages et ma thèse, vous n'avez cessé de m'accompagner, me guider, me faire grandir et me faire découvrir un monde alors inconnu. J'espère que ce n'est que le début d'une longue collaboration.

Merci également à **Pascal Lafourcade** et **Jean-Marc Vincent** qui m'ont guidée dans cette voie.

Merci à **Alain Cournier**, **Ajoy K. Datta**, **Franck Petit** et **Lawrence L. Larmore** pour toutes ces discussions très intéressantes. J'espère que nous aurons d'autres occasions de travailler ensemble dans le futur. Merci également à Ajoy et Lawrence de m'avoir accueillie à Las Vegas pour un séjour court mais riche en discussions.

Je veux également remercier **Christian Scheideler** et **Sébastien Tixeuil** d'avoir accepté de rapporter ma thèse, ainsi que **Paola Flocchini**, **Pierre Fraigniaud**, **Colette Johnen** et **Michel Raynal** d'avoir accepté de faire partie de mon jury. Vos remarques sur mon travail ouvrent à de nombreuses réflexions.

Je remercie mes co-bureaux qui m'ont supportée pendant ces trois années, **Alexandre**, **Alexis**, **Moustafa** et **Valentin**. Merci aux joueurs de coinche, **Amaury**, **Denis** et **Guillaume**, pour ces pauses déjeuner toujours très animées. Merci à tous mes autres

collègues de Verimag, en particulier **Cristina**, **Dinh**, **Hamza**, **Josselin**, **Louis** et **Yuliia**.

Merci à mes collègues de l'Ensimag avec qui j'ai eu le plaisir d'enseigner pendant deux années. Je remercie en particulier **Grégory Mounié**, **Marie-Laure Potet** et **Claudia Roncancio** qui ont eu la patience de répondre à toutes mes questions sur les enseignements.

Merci à tous mes amis qui ont réussi à me faire sortir le nez de ma thèse pendant quelques heures, **Alexandre**, **Carole-Anne**, **Céline**, **Julie** et **Pierre**.

Je voudrais enfin remercier **Maxime** d'être à mes côtés et de m'avoir soutenue pendant cette thèse malgré la difficulté d'être tous les deux doctorants.

Même si je ne pourrai jamais les remercier assez pour tout ce qu'ils ont fait pour moi, merci à mes parents d'être toujours là quand j'en ai besoin, de m'accompagner et me soutenir. Je n'en serais pas là aujourd'hui sans vous.

Contents

Chapter 1

Introduction

"Begin at the beginning," the King said gravely, "and go on till you come to the end: then stop."

— Lewis Carroll, *Alice's Adventures in Wonderland*

Contents

When Jane wakes up, motion sensors detect her awakening and switch on the lights with a progressive brightness and heating in the bathroom. During Jane's ride to work, her GPS informs her of a road traffic accident signaled by other users. She can adapt her route to avoid the resulting traffic jam. When she arrives at work, she quickly finds a free parking spot thanks to the sensors that monitor the parking lot occupation. Throughout her workday, Jane exchanges emails with her clients on the other side of the world. She takes part in a video conference meeting with another branch and exchanges data with her colleagues through the local network of the company. While she is away, the sensors of her solar panels detect high output at midday and switches on the hot water tank and the dishwasher. When she comes back home, Jane checks the last news on Internet, before sharing the photos of her last weekend with her family through a cloud file storing service.

Every situation described in the above example involves a distributed system. Those systems are ubiquitous and unavoidable in our everyday life. With an increasing number of users, such distributed systems are becoming more and more wide and complex. Thus, we need efficient algorithms to make these systems work. Moreover, distributed systems are very diversified and can be used in many varied context such as houses, streets, or factories as shown in the above example, but also in even more adversarial environments (*e.g.*, wireless sensor networks deployed in a desert or around a volcano). However, these contexts may be uncertain, *i.e.*, the context is not fully known *a priori* or is unsettled. For instance, wide systems composed of cheap mass-produced devices are highly exposed to dysfunctions and crashes. These dysfunctions and crashes cannot be foreseen, yet the service provided by the distributed system must always remain available. Another example, the nature itself of systems may be highly dynamic, *e.g.*, mobile networks. A mobile phone user can move around and change of relay mast during a phone call, yet the call should not be interrupted. Thus, the distributed systems must be resilient to uncertainty. The development of large-scale social networks, where huge amounts of data circulate over the world, is coupled with the increasing need of privacy. This need shows that, in some cases, uncertainty is not a drawback, but rather a requirement (of the user). The privacy concern has justified the design of solutions for anonymous networks. Partial anonymity can be also obtained in homonymous networks, where identifiers are not necessarily unique. The need of privacy is usually considered as a security requirement. Despite security is out of the scope of this thesis, we will nonetheless study various levels of anonymity in our solutions.

1.1 Distributed Systems

In computer science, a *distributed system* [Tel00, Lyn96] is any computational application where several computers or processors cooperate to achieve some common goal. More precisely, a distributed system is a set of autonomous yet interconnected computational units. A *computational unit* is a computer, a core of a multicore processor, a process in a multitask operating system, *etc.* For sake of simplicity, computers, processors, and processes will be referred as *processes* in the following. Those processes can be geographically spread. *Autonomous* means that each process has its own control. It does not rely on some central controller. *Interconnected* means that processes are able to exchange information, directly or indirectly, *e.g.*, sending messages through wires or radio-waves, through shared memories. This definition includes parallel computers, computer networks, sensor networks, mobile ad hoc networks (MANETs), robot fleets, *etc.*

1.1.1 Characteristics and Differences with Central Systems

Distributed systems are often defined in opposition to central systems. Indeed, distributed systems have particular characteristics:

- **No Global Time:** In distributed systems, the speed of computation of each process is heterogeneous and the communication are usually asynchronous. The processes cannot rely on a global clock. In particular, their local clocks may drift. Hence, contrary to centralized systems, the actions of processes may not always be ordered. We can only rely on a *causal order* [Lam78]. For example, on Figure 1.1, the sending

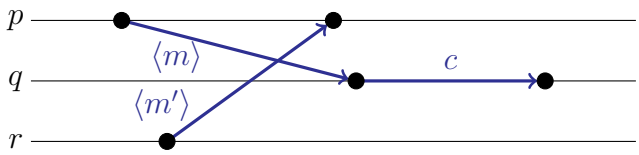


Figure 1.1 – Causal order of events.

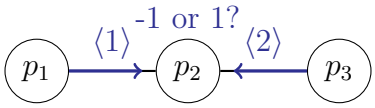


Figure 1.2 – Example of non determinism.

of a message m by p is before the reception of m by q, the local computation c of q is after the reception of m, but the sending of m by p and the sending of m' by r are *independent* (or *concurrent*), *i.e.*, from the point of view of a process, it is impossible to distinguish if the sending of m happens before or after the sending of m'.

- **No Global Knowledge:** Contrary to centralized systems, where decisions are made according to the global state of the system, processes of a distributed systems must rely on their local knowledge, *i.e.*, their local memory, to decide their next actions. In particular, even if the local memory of a process can be updated according to received information, this information may be outdated due to the asynchronism of the system.

- **Non-determinism:** Due to the asynchronism of processes and communications, the execution of a deterministic distributed algorithm may lead to different results and this result is not always predictable, while the execution of a deterministic sequential algorithm depends only on its inputs. For example, on Figure 1.2, p_1 sends value 1 to p_2, p_3 sends value 2 to p_2, and p_2 computes the subtraction of the first received value by the second received value. If p_2 receives 1 first, the computed result is -1, otherwise, the computed result is 1. Thus, a distinction has been made between a *function* and a *task* [MW87]. More precisely, contrary to a function that associates only one output vector (*i.e.*, the vector of the outputs of each process) to each possible input vector (*i.e.*, the vector of the inputs of each process), a task associates a set of possible output vectors to each input vector.

1.1.2 Examples of Motivations and Applications

Distributed systems have a lot of applications and are ubiquitous in our everyday life. Depending on the application, distributed systems may simply be necessary or may be preferred over sequential and central systems for various reasons. Non-exhaustive examples are exposed below.

Simplify Communications. In 1969, a *wide-area network (WAN)* called ARPANET is created between major American universities to facilitate the cooperation and exchange

of data between these organizations. ARPANET is the ancestor of Internet, that connects billions of computers and other devices.

Nowadays, our communications rely mainly on distributed systems: emails, voice over IP (VoIP) technologies (*e.g.*, Skype, Google Talk, Discord), instant messaging applications (*e.g.*, WhatsApp, Yahoo!Messenger, Google Hangouts), peer-to-peer (P2P) file sharing networks (*e.g.*, Gnutella, eDonkey), *etc.*

Faster and Remote Computations. By multiplying the processes, the computation of some long task may be split among several processes, resulting in a speed up of the computation. That is the objective of parallel computers. For example, IBM supercomputer Deep Blue was designed to compute fast chess moves. But geographically spread networks can also be used for distributed computing. For example, in volunteer computing projects, everyone can give some computational power or storage of its personal computer to help at the computation of some hard task, *e.g.*, search for extraterrestrial radio transmission in SETI@Home project, analyze the structure of proteins for medical research in Rosetta@Home project.

To facilitate computing on remote distributed networks, a lot of companies propose cloud services, *e.g.*, Amazon Elastic Compute Cloud, Microsoft Azure. Cloud computing provides on-demand access to shared computational power and storage. Notice that some cloud services are dedicated to file hosting, *e.g.*, DropBox, Google Drive.

Monitoring. *Wireless sensor networks (WSNs)* are composed of numerous sensors generating data about their environment. Those sensors are equipped of wireless communication abilities. WSNs can be used to monitor natural disasters, *e.g.*, volcanic eruptions, earthquakes. Their usage is also gradually increasing in emerging technologies of home automation and smart cities to monitor power consumption, lighting, *etc.* Swarm of drones and robot fleets can also be used to monitor an area and for military applications.

Increase Availability and Resiliency. By duplicating the number of processes executing the same task, the availability of a service is improved against potential failure of a process. Notice that computational replication requires an arbitration between the results of the different replicated processes. A similar technique can be used to improve the availability of data, by replicating them on several storage disks. In particular, data replication can be made on geographically distant data servers to improve resiliency.

Sharing Resources. As stated before, distributed systems allow to share data, computational power, storage disks, *etc.* It may also be needed to share other peripherals, *e.g.*, printers among the employees of a company, since these devices are expensive. Usually, the number of shared resources is far smaller than the number of processes. Thus, every processes cannot access to the resource they required at the same time, and we must manage a fair access to resources.

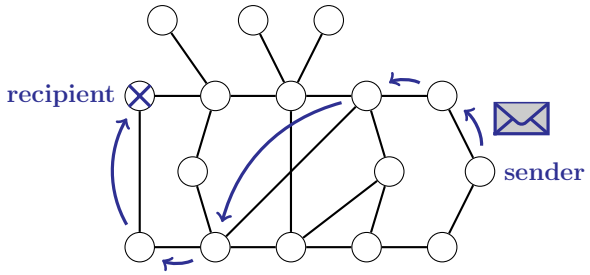


Figure 1.3 – Example of routing.

1.2 Computation in Distributed Systems

Processes of a distributed system aim to fulfill a global task using their local inputs.

1.2.1 Classical Problems in Distributed Computing

Due to the characteristics of distributed computing, the design of distributed algorithms requires to face fundamental problems in order to solve higher-level distributed tasks. Some examples are listed below.

Routing. A process is not necessarily directly connected to every other process. Hence, when it needs to send information to another process, it does so indirectly. The information goes from process to process along some path until reaching the destination, see an example on Figure 1.3.

- **Routing:** The *routing* problem consists in building a *routing table* at each process, *i.e.*, for every possible recipient process, to which directly accessible process the information should be sent. Notice that this table may change over time, in particular when communications abilities are dynamic.

- **Broadcasting:** The *broadcasting* problem consists in dispatching some information to every process. The difficulty is to prevent an infinite circulation of the information in the network.

- **Propagation of Information with Feedback:** When a process does not only need to send information to every process but also to get back some response, it needs to do a *propagation of information with feedback (PIF)* [Cha82, Seg83]. The response received by the initiator of the PIF is an aggregate of the responses of every other processes.

Agreement. Since there is no central control, processes may require to decide and agree on some information.

- **(Binary) Consensus:** In the *(binary) consensus* problem, each process initially propose a Boolean value, and the processes must agree on a single value among those proposed. This decision must be irrevocable, *i.e.*, processes cannot change their decision afterwards, and every process must decide the same value.

- **k-set Agreement:** Numerous variant of the consensus problem have been defined. For example, the *k-set agreement* [Cha93] is a variant of the consensus with weaker conditions, *i.e.*, processes may not agree on the same value, as long as there is no more than $k \geq 1$ different decided values.

- **Leader Election:** The *leader election* problem [Lan77] consists in distinguishing a unique process as the *leader*. This problem is fundamental in distributed computing, since it allows the designed leader to make decisions for the whole system, and so to ensure central control. We study this problem in Chapters 3 and 4.

Resource Allocation. When resources are shared among several processes, *e.g.*, a printer shared between employees of a company, you want to be sure that a process needing the resource will be able to eventually access it, *e.g.*, nobody monopolizes the printer, and you do not want conflicts when accessing the resources, *e.g.*, two files are not printed at the same time. Resource allocation problems consist then in managing a fair access to resources. Some examples are given below. We study resource allocation in Chapter 6.

- **Mutual Exclusion:** The *mutual exclusion* problem [Dij65, Lam74] is the simplest resource allocation problem. Only one resource is shared among every processes and at most one of them can execute its *critical section*, *i.e.*, use the resource, at a time. There are two approaches to mutual exclusion. In the *request-based* approach, a central controller manages the requests of processes to access resources. On the contrary, in the *token-based* approach, one token circulate among processes and a process can enter critical section only when it holds the token.

- **ℓ-exclusion:** In the *ℓ-exclusion* problem [FLBB79], $\ell \geq 1$ copies of a reusable resource are shared among processes. Thus, at most ℓ processes can concurrently execute their critical section.

- **k-out-of-ℓ Exclusion:** The *k-out-of-ℓ exclusion* problem [Ray91] is a generalization of the ℓ-exclusion problem where processes can request and use up to k resources, $1 \leq k \leq \ell$. A direct application of this problem is the management of the bandwidth, *i.e.*, the total bandwidth cannot exceed ℓ units, while each process is allowed to use up to some quota k.

- **Dining Philosophers and Local Mutual Exclusion:** In the *dining philosophers* problem [Dij78], philosophers sit around a round table. Each philosopher has a fork at his left and at his right, that he shares with its left and right neighbor, respectively. Every philosopher wants to eat but to do so it must use both its left and right fork, and then he prevent its neighbors to eat at the same time. The generalization of this problem is the *local mutual exclusion* problem, where two neighboring processes cannot execute their critical section at the same time.

- **Group Mutual Exclusion:** In *group mutual exclusion* problem [Jou98], every process requesting the same resource can use it concurrently, but two processes requesting different resources cannot execute their critical section at the same time. The management of a CD player is a good illustration of this problem: when one

CD is played, everyone in the room can listen to it, yet if someone wants to listen to another CD, it cannot do so at the same time.

Building Spanning Structures. The topology of a distributed system, *i.e.*, the communication links between processes, may not be organized. Nonetheless, solving some problems is easier and/or faster when the system has a certain structure, *e.g.*, doing a broadcast from the root of a tree. Thus, building spanning structures is a fundamental problem of distributed computing. Most of the problems cited below were defined in graph theory, but must be solved with the additional difficulty of the distributed computation.

- **Spanning Trees:** Building a spanning tree over the network is one of the most studied problems of distributed computing. It may be required that the resulting tree has some properties. *Breadth First-Search (BFS) spanning trees* [Moo57] minimize the distance between the root and the other processes. *Minimum spanning trees (MST)* [Bor26, Kru56, Pri57] minimize the sum of the weights of communications links in the resulting tree.

- **Clustering:** The *clustering* problem [BEF84] consists in partitioning the network into clusters. Each cluster is a connected subset of processes with one of them distinguished as clusterhead. Clustering is often used to design efficient communication mechanism. Processes that are not clusterhead can communicate with the other processes inside the same cluster. Communications between clusters are managed by clusterheads. Some variants of the clustering problem have been defined, *e.g.*, in the *k-clustering* problem [APHV00], every process inside a cluster is distant of at most $k \geq 0$ hops from the clusterhead.

Coloring. Sometimes, we need to locally differentiate processes by giving them a *color*. Again, the examples of problems listed below come from graph theory field.

- **(Vertex) Coloring:** The *(vertex) coloring* problem consists in giving a color to each process such that no two neighbors have the same color. On of the objectives is to use as less different colors as possible.

- **Distance-k Coloring:** In the *distance-k (vertex) coloring*, two processes that are distant of up to $k \geq 1$ hops cannot have the same color. Thus, the vertex coloring problem is equivalent to the distance-1 vertex coloring problem.

- **Edge Coloring:** In the *edge coloring* problem, instead of coloring processes, we give colors to the communication links such that two communication links of the same process do not have the same color.

Synchronization. As stated before, communications and processes are typically asynchronous in a distributed system. Nonetheless, it is easier to design algorithms for synchronous systems, since there is less non-determinism. Furthermore, it is impossible to deterministically solve some problems without hypotheses on the synchrony, *e.g.*, deterministic consensus if one process may crash [FLP85].

- **Synchronizer:** The idea of a *synchronizer* [Awe85] is to simulate pulses, *i.e.*, phases of execution where every process sends messages (possibly zero), then receives messages (possibly zero), then realizes some local computation. In particular, a synchronizer certifies that messages sent during a pulse are received during the same pulse.

- **Phase Synchronization/Barrier Synchronization:** In the *phase synchronization* problem [Mis91], every process holds a (bounded or unbounded) local clock. The objective is to synchronize those clocks such that a process cannot increment its clock to $c+1$ until the clock of every other process gets value $c \geq 0$. Moreover, each process must increment its clock infinitely often. This problem is also called *barrier synchronization*.

- **Asynchronous Unison:** The *(asynchronous) unison* [CFG92] is a weaker variant of phase synchronization: clocks must be synchronized such that the difference between the clocks of two neighboring processes is at most one. This problem and some of its variants are studied in Chapter 5.

1.2.2 Performances

The size of distributed systems increases with the democratization of connected devices. For example, the number of Internet users in the world moves from one billion users in 2005 (approximately 16% of the worldwide population) to 3.5 billion in 2016 (approximately 47%). With the growth of distributed systems, their complexity also increases. Thus, to maintain the usefulness of distributed systems, the designed distributed algorithms must be efficient.

First, the computation should be fast and the provided service must always be available. In addition, distributed systems contain more and more embedded systems, *e.g.*, wireless sensors, which have limited resources (small battery, small computation power, small memory). Thus, the complexity in memory, the number of exchanged messaged, and the complexity of the computation itself should be small. Otherwise, the processes might not be able to execute their algorithm at all, or might drain their battery.

1.2.3 Uncertain Context

In this thesis, uncertain context means that the context of execution of the distributed system is not fully known *a priori* or is unsettled. In particular, we focus on non fully identified systems where faults can occur.

On the contrary, if no fault hits the system, *i.e.*, the system continuously satisfies its specification, and if processes are identified, *i.e.*, every process has a unique identifier (ID), most of problems that can be solved in a central system (in particular, static problems[1]), can also be solved in a distributed system.

[1] A *static* problem is a problem where the expected computation is finite and returns a result according to the inputs, *e.g.*, building a spanning tree, electing a leader.

For example, to compute a spanning tree, processes can elect a *leader*, *i.e.*, a unique distinguished process. This leader can execute a snapshot to collect the local states, and in particular the inputs of the problem, of every other process. Thus, the leader can be aware of the whole topology and inputs of the system and can centrally compute the result, *i.e.*, the spanning tree, in a central way before broadcasting it to the other processes.

This technique is very costly and cannot be applied for real systems. Indeed, it requires a large amount of memory at the leader, a lot of exchanged messages, and a long computation time. Obviously, there exists far more efficient algorithms to build a spanning tree and a part of research in distributed computing focuses on designing efficient distributed algorithms under these conditions. Nonetheless, this technique shows the feasibility of problems in distributed systems.

Absence of Identifiers. Because of the size and complexity of distributed systems, assuming that processes are identified may be unrealistic, in particular for cheap and massively produced and/or deployed devices. Moreover, even when processes are identified, one may not want to publicly communicate its ID for security or privacy reasons.

However, in an *anonymous* network where processes do not have IDs, many fundamental problems become impossible to solve. In particular, it is impossible to deterministically break symmetries of the network topology. For example, the leader election problem cannot be deterministically solved in an anonymous network since two processes cannot be distinguished except by their inputs and their degree, *i.e.*, the number of processes with whom they can directly communicate. (In particular, every process has the same degree if the topology of the system is regular.) Yamashita and Kakugawa propose a survey of computable problems in anonymous networks in [YK96].

To circumvent these impossibility results, there are two main approaches. First, one can provides probabilistic solutions. For instance, if two neighboring processes cannot be distinguished, they can "flip a coin" until getting a different result. However, with this solution, the specification of the considered problem is only ensured with some probability. On the other hand, the second approach consists in considering in-between models of anonymity, neither (fully) identified (*e.g.*, processes have a unique ID), nor (fully) anonymous (*e.g.*, processes do not have IDs). For instance, we can consider the *homonym* processes model [YK89] where processes have IDs, but these IDs may not be unique. In this case, processes with the same identifier are called *homonyms*.

Presence of Faults. When the size of a distributed system increases, it becomes more exposed to the failure of some process. Indeed, processes may crash, their memory may be corrupted, *etc.* Moreover, the devices that composed distributed systems are often produced on a large scale and cut-rate, thus they are more vulnerable. Finally, wireless communications are increasingly used, while they are more vulnerable. In 2016, the number of "things" connected to Internet was estimated at 7 billion. If we add computers, smartphones, and tablets, we reach the number of 18 billion of connected devices. In distributed systems of such a size, it is impossible to assume that no fault will occur, even during only a couple of hours.

Now, as explained before, distributed systems are ubiquitous in our everyday life and people are increasingly dependent of them. If a disruption of service, even temporary, would hit such a system, the consequences would be severe.

Nonetheless, due to the complexity, the extent, and/or the usage of distributed systems, ensuring a human maintenance is often too complicated, too slow, or even too dangerous. Hence, distributed systems should be resilient to faults. In the next section, we define and detail the considered faults and study the resiliency of distributed systems against faults.

1.3 Fault Tolerance

In computer science, we say that a *fault* leads the system to an *error* that causes a *failure*. A component or system suffer from a failure when its behavior is not correct w.r.t. its specification. An error is a state of the system that may lead to a failure. It can be a *software error*, *e.g.*, division by zero, non-initialized pointer, or a *physical error*, *e.g.*, disconnected wire, turned off CPU, wireless connection drop. A fault is an event leading to an error, *i.e.*, a programming fault leading to a software error or a physical event (*e.g.*, power outage, disturbance in the environment of the system) leading to a physical error. In this thesis, we consider only physical errors.

1.3.1 Fault Classification

The different kind of faults can be classified according to:

- their localization: whether the component hit by the fault is a communication link or a process.
- their origin: whether the fault is *benign*, *i.e.*, due to physical problems, or *malign*, *i.e.*, due to malicious attacks.
- their duration: whether the fault is *permanent*, *i.e.*, longer than the remaining execution time (*e.g.*, a crash), *transient*, or *intermittent*. There is a slight difference between transient and intermittent faults. On average, during an execution, a transient fault hits the system once, while an intermittent fault hits the system several times.
- their detection: whether a process can detect according to its local state when it is hit by a fault.

Some examples of faults are listed below:

- *Crash:* Process that definitively stops to execute its algorithm. *Initially dead processes* is a subcategory of crashes, *i.e.*, processes that does not execute any computational step.
- *Byzantine:* Process with arbitrary behavior. In particular, it may not execute correctly with respect to its algorithm, *e.g.*, virus.
- *Intermittent Loss of Messages:* Communication link that frequently looses messages.

- *Transient faults:* Component that temporarily presents a faulty behavior but this fault does not lead to a permanent damage of the hardware. After the end of transient faults, the state of hit components may be corrupted, *e.g.*, local memories corruption, messages corruption.

1.3.2 Achieving Fault-tolerance

Without faults, it is possible to solve in distributed computing (almost) everything that is possible to solve in sequential computing providing that processes are identified or there is a leader. (Notice that both assumptions are equivalent since it is possible to elect a leader in an identified network and it is possible to name processes if there is a leader.) Nonetheless, faults must be considered to increase the resiliency of distributed systems.

Now, Fischer et al. showed in [FLP85] that, in presence of crashes, it is impossible to deterministically solve the consensus problem in asynchronous systems, when there is at most one crash. This result holds despite the network is complete (*i.e.*, each process can directly communicate with every other processes), and no message is ever lost. This impossibility results can be extended to a large class of fundamental problems of distributed computing [MW87], *e.g.*, atomic broadcast [CT96].

Several approaches are used to circumvent this impossibility. There are two main solutions, either assuming additional hypotheses or weakening the specification of the problems.

One can assume a *failure detector*, *i.e.*, an oracle that informs processes of failures that previously hit the system. For example, the *perfect failure detector* [CT91], denoted $\mathcal{P}$, eventually informs every non-faulty processes of every failure that actually happened, and does not suspect any non-faulty process. It is also possible to restrain the number of faults, *e.g.*, in [CHT96], Chandra et al. solves the consensus problem assuming that a majority of processes are correct and assuming the *eventually weak failure detector*, denoted $\diamond\mathcal{W}$, such that, after a while, every faulty process is always suspected by at least one correct process, and every correct process is no more suspected by other correct processes. Finally, it is possible to make assumptions on the synchrony of processes, *e.g.*, in [DLS88], Dwork et al. solves the consensus problem assuming that the difference of speed between two processes is bounded.

On the other hand, there are two main approaches to weaken the specification of the considered problem. First, one can provide a probabilistic solution to the problem, *e.g.*, the probabilistic algorithm for consensus that withstand crashes of Ben-Or [Ben83]. In this latter algorithm, the liveness of consensus is ensured with probability 1. The second approach is the design of algorithms that satisfy specifications where the safety is relaxed, *i.e.*, stabilizing algorithms [Dij74].

Robust *vs.* Stabilizing Algorithms. To sum up, two approaches to design resilient distributed systems have been studied: a pessimistic approach, *i.e.*, designing robust algorithms, and an optimistic approach, *i.e.*, designing stabilizing algorithms.

In a *robust algorithm*, every received information will be suspected in order to guar-

antee the correct behavior of non-faulty processes. Strategies such as voting to consider certain information only if enough other processes claimed the receipt of similar information are used in these algorithms. Thus, robust algorithms withstand permanent faults and must be considered when even a temporal interruption of service is unacceptable.

On the contrary, when short and rare interruptions of service can be accepted (short and rare compare to the overall availability of the service), *stabilizing algorithms* offer a lightweight approach to withstand transient faults. Indeed, after the end of transient faults, the behavior of the system, even of non-faulty processes, may be incorrect. Nonetheless, stabilizing algorithms ensure a convergence in finite time to a correct behavior, as long as the time between two transient faults periods is longer than the recovery time. Self-stabilization and its variants presented in Subsection 1.4 are examples of this approach.

Notice that some algorithms are both robust and stabilizing, *e.g.*, [GP93].

1.4 Self-stabilization and Variants

Self-stabilization [Dij74, Dij86] is a versatile approach that allows distributed systems to withstand transient faults. After the end of transient faults, the system may be in an arbitrary configuration. If the system is self-stabilizing, it recovers a correct behavior in finite time, without any external help (in particular, any human intervention). The configurations where the system has a correct behavior are called *legitimate configurations*. Notice that the recovery does not depend on the nature (*i.e.*, if the faults hit processes and/or communications links) nor the extent (*i.e.*, how many components are hit), with the exception of modifications of the code.

Nonetheless, this versatility has two main drawbacks. First, the specification of the system is not ensured during its recovery, *i.e.*, there is no safety guarantee during the recovery. Moreover, the processes are not able to locally detect the end of the recovery. Thus, it is not possible to ensure the termination detection.

We propose a self-stabilizing algorithm in Chapter 4.

1.4.1 Variants of Self-stabilization

Self-stabilization led to numerous variants. A non-exhaustive list of related properties is exposed below.

Stronger Variants. To counter the drawbacks of self-stabilization, some variants ensuring stronger guarantees have been proposed.

Safe convergence [KM06] ensures safety guarantees during the recovery. More precisely, after the end of transient faults, a safely converging self-stabilizing algorithm converges quickly, *i.e.*, it is usually required to converge in $O(1)$ round, to a so-called feasible configuration, where a minimum quality of service is ensured. Then, it converges (more slowly) to an optimal configuration, where the (full) specification of the system is reached. Safe-convergence is mainly used for the computation of optimized structure, *e.g.*, in [KM06], Kakugawa and Masuzawa propose a safely converging self-stabilizing

algorithm that quickly builds a dominating set, and then converges to a minimal dominating set. *Snap-stabilization* [BDPV07] ensures even stronger guarantees. Indeed, a snap-stabilizing algorithm recovers immediately after the end of transient faults. We propose a snap-stabilizing algorithm in Chapter 6.

Some variants have been defined to converge faster depending on the extent and/or the nature of the faults. *Fault-containment* [GGHP96] ensures that, when only a small number of entities are hit by transient faults, the incorrect behavior is contained within a determined radius around faulty components. This allows a quicker convergence. Similarly, when $k \geq 0$ components are hit by transient faults, a *time-adaptative* algorithm [KP99] converges in $O(k)$ time units. Finally, *superstabilization* [DH97] is a variant of self-stabilization defined especially for dynamic networks, *i.e.*, networks where processes may leave or enter the system and communication abilities may change over time. After a unique topological change, a superstabilizing algorithm recovers very quickly its correct behavior. Furthermore, a passage predicate is guaranteed during the convergence.

Notice that we propose a variant of superstabilization and safe convergence in Chapter 5.

Weaker Variants. Some variants of self-stabilization ensuring weaker guarantees have also been defined. For example, a *k-stabilizing* algorithm [BGK98], $k \geq 1$, converges in a self-stabilizing way provided that there is no more than k faulty processes. More precisely, we consider a *Hamming distance* [DH97] between configurations, *i.e.*, the number of processes whose state is different in the two considered configurations. Then, a k-stabilizing algorithms converges if the minimum Hamming distance between the initial configuration and a legitimate configuration is k.

The difference between self-stabilization and the two next variants is more subtle. Self-stabilization ensures that the system converges in finite time to a correct behavior, while every execution a *pseudo-stabilizing* algorithm [BGM93] contains a suffix where the system has a correct behavior, however we cannot bound the time needed to ensure convergence. Self-stabilization ensures that, every execution starting from a given incorrect state (resulting of transient faults) converges to a correct state. On the contrary, *weak stabilization* [Gou01] ensures that at least one execution starting from this incorrect state converges.

Finally, notice that every property previously exposed is defined for deterministic algorithms. Nonetheless, some probabilistic variants were also proposed, *e.g.*, *probabilistic self-stabilization* [IJ90].

1.4.2 Expressiveness and Limitations of Self-stabilization

As previously exposed in Section 1.3.2, it is not always possible to solve problems in a fault-tolerant context. Hence, the expressiveness of self-stabilization has been extensively studied. In [KP93], Katz and Perry proposed a protocol that transforms almost every non-stabilizing algorithm written in the message-passing model (*i.e.*, a computational model where processes communicate by exchanging messages) into a self-stabilizing one. More precisely, Katz and Perry showed that their transformer works for any problem whose

specification is *suffix-closed*.[2] They also showed that this condition is necessary. The principle of their transformer is to let execute the non-stabilizing algorithm concurrently with a self-stabilizing snapshot algorithm *i.e.*, a protocol that creates a copy of the state of the entire system at a process. The snapshot algorihm regularly controls if the system is in a legitimate configuration. If the snapshot protocol detects that the system is in an illegitimate configuration, then a full reset of the network is made, using a self-stabilizing reset algorithm. The snapshot and the reset protocols are based on a self-stabilizing PIF algorithm. Hence, whenever the self-stabilizing PIF algorithm can be designed a given model, then this self-stabilizing contruction holds and the result applies. However, notice that the purpose of this construction is only to demonstrate the feasibility of transforming almost any algorithm to a corresponding self-stabilizing algorithm. As a consequence, the method, although very general, is clearly inefficient.

Despite this transformer originally requires unbounded process memories (which is not feasible in real systems) to tackle the unbounded process memories capacity assumption on links, it can be applied in other models using finite process memories. For example, if we consider a message passing model with links of bounded capacity, it is possible to use the self-stabilizing PIF protocol of Varguese [Var00] to design the transformer.

Notice that the transformer of Katz and Perry requires that there is a distinguished process (the one that executes the snapshots and resets). This process can be computed by a self-stabilizing leader election algorithm, *e.g.*, [ACD+16], provided that the processes are identified. If the processes are not identified, the impossibility results on anonymous networks (see Section 1.2.3) remains true for self-stabilizing solutions.

Most of self-stabilizing algorithms are designed in the locally shared memory model, *i.e.*, a computational model where processes communicate through shared memories. Designing self-stabilizing algorithm using lower level communication models such as asynchronous message passing is more challenging. The key point is that we aim to design self-stabilizing solutions that only require a bounded memory per process, since an unbounded memory would not be feasible in real systems.

Gouda and Multari showed in [GM91] that designing deterministic self-stabilizing algorithms with bounded memory is impossible for a large class of problems if the communications links are not *bounded*, *i.e.*, processes do not know how many messages can transit by a link at a time. This class of problems includes the *alternating bit protocol (ABP)*, *i.e.*, a communication protocol that withstand message loss. (Notice that these results assume FIFO links. Nonetheless, they can be extended to non-FIFO links using the results of Dolev et al. in [DDPT11].) Afek and Brown [AB93] proposed a probabilistic self-stabilizing ABP that does not require an unbounded memory but an infinite sequence of random numbers. Nonetheless, we focus here on deterministic solutions. The ABP problem is fundamental since it allows to remove faulty messages of the communication links (those that were inside the links initially, and those sent due to reception of faulty messages).

[2] We say that a specification SP is *suffix-closed* if there exists an assertion A in (future) linear temporal logic such that for every execution e, e satisfies SP if and only if A is TRUE in the terminal configuration of e, if e is finite, or A is infinitely often TRUE in e, otherwise.

Thus, to obtain a self-stabilizing algorithm with bounded memory, most problems requires that processes *a priori* know a bound on the capacity of communication links, *e.g.*, [ABB98, HNM99, Var00, AN05, AKM+07]. Nonetheless, for more restricted class of problems, this assumption is not necessary. For example, in [APV91], Awerbuch et al. showed that there exists self-stabilizing solutions for a class of problems, called *locally correctable* problems, that requires only bounded memory without requiring bounded communication links. Similarly, Delaët et al. [DDT06] have proved that it is possible to design silent self-stabilizing algorithms using bounded memory for a class of fix-point problems even if the communication links are unreliable and their capacity is unbounded.

1.5 Contributions

In this thesis, we study several classical problems of distributed computing under uncertain contexts and we explore both research axes previously exposed: going towards more anonymity by proposing efficient algorithms for networks with anonymous or homonym processes, and ensuring greater fault tolerance by proposing self-stabilizing algorithms ensuring increasing safety guarantees. Notice that we focus on deterministic solutions.

Chapter 2 detailed the computational models used in this thesis. More precisely, we formally define distributed systems, and we introduce the message passing and the locally shared memory models.

Then, we present the contributions.

- **Chapter 3 (Leader Election in Unidirectional Rings with Homonym Processes):** In Chapter 3, we present results from [ADD+16a, ADD+17a, Dur17]. We study the leader election problem in the model of homonym processes, *i.e.*, the identifiers of processes may not be unique, in between the (fully) identified and the (fully) anonymous model. We focus on unidirectional rings networks and we study in which classes of unidirectional rings the problem can be solved. More precisely, we show that it is impossible to solve leader election in rings in four different class of unidirectional rings: rings with symmetric labeling, rings that contains at least one unique label (class denoted here $\mathcal{U}^*$), rings with asymmetric labeling (denoted $\mathcal{A}$), and in rings that contains up to $k \geq 1$ processes with the same label (denoted $\mathcal{K}_k$). Then, we propose a leader election algorithm for class $\mathcal{U}^* \cap \mathcal{K}_k$ and two algorithms for $\mathcal{A} \cap \mathcal{K}_k$.

- **Chapter 4 (Self-stabilizing Leader Election under Unfair Daemon):** Chapter 4 summarizes the results of [ACD+14, ACD+15, ACD+16]. Similarly to Chapter 3, we study the leader election problem but in a different context. Indeed, we focus there on solving the leader election in identified networks of arbitrary connected topology under the distributed unfair daemon, the more general scheduling assumption of the model. We aim to design silent and self-stabilizing algorithms for this problem that require no knowledge on the network. More precisely, we propose the first algorithm working under such assumptions that stabilizes in a polynomial number of computational steps, and we show that the previous best algorithms of literature converge in a non-polynomial number of steps.

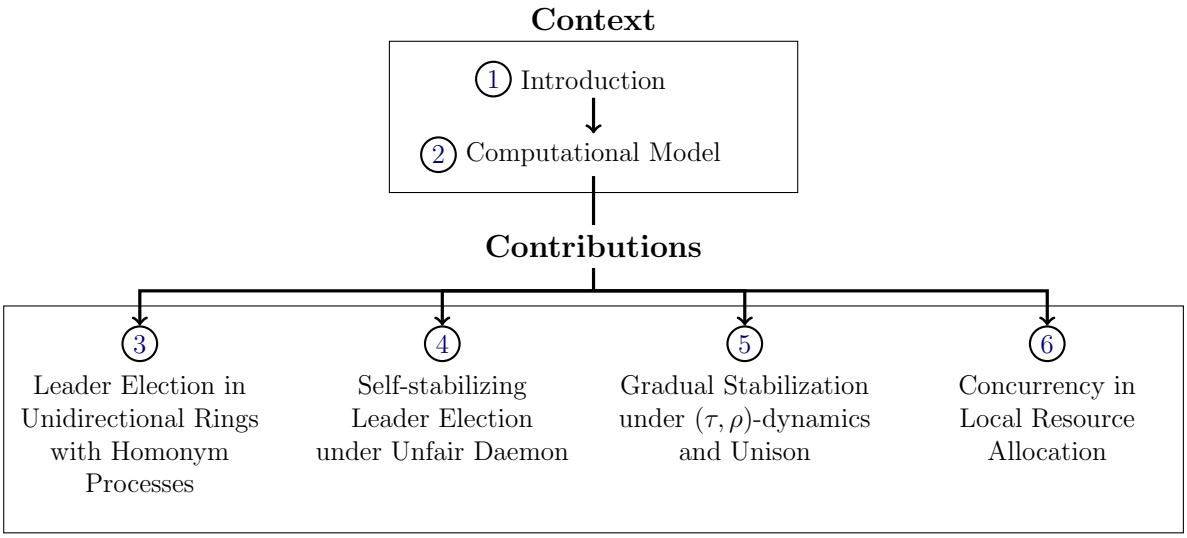


Figure 1.4 – Roadmap.

- **Chapter 5 (Gradual Stabilization under (τ, ρ)-dynamics and Unison):** The results published in [ADDP16] are presented in Chapter 5. We propose a variant of self-stabilization, called gradual stabilization, designed especially for dynamic networks. Indeed, a (τ, ρ)-gradually stabilizing algorithm ensures fast convergences after up to $\tau \geq 1$ ρ-dynamic steps (*i.e.*, steps containing topological changes that satisfy predicate ρ) hit the system. We illustrate this new property by proposing the first self-stabilizing unison algorithm designed for dynamic networks.

- **Chapter 6 (Concurrency in Local Resource Allocation):** Finally, in Chapter 6, we present the results of [ADD15, ADD16b, ADD17b]. We study there the question of concurrency in resource allocation problems. We propose a versatile property that allows to express concurrency in any resource allocation problem. We illustrate this property by studying a large class of problems, so-called local resource allocation (LRA). We show that the higher level of concurrency, called maximal-concurrency, cannot be achieved without violating the fairness of LRA. Thus, we propose a partial concurrent LRA algorithm, that ensures a high (yet not maximal) degree of concurrency. This algorithm is additionally snap-stabilizing.

Roadmap. Each contribution chapter is independent. Nonetheless, a generalization of the computational models used in this thesis is presented in Chapter 2. Thus, we incite readers to read Chapter 2 before reading a contribution chapter. Figure 1.4 illustrates the dependencies between chapter.

Notice that the contributions of the thesis are organized by increasing safety guarantees order, but it can be read in different orders. Some reading guides are described bellow.

- **Safety guarantees (Figure 1.5):** We propose algorithms that provide increasing safety guarantees. In Chapter 3, we propose algorithms that are not stabilizing, *i.e.*, we assume a particular initial state. In Chapter 4, we design and study self-

stabilizing algorithms. Then, in Chapter 5, we propose a variant of self-stabilizing that offers additional safety guarantees during the convergence in case of topological changes. Finally, in Chapter 6, we propose a snap-stabilizing algorithm, *i.e.*, the system immediately recovers a correct behavior after the end of transient faults.

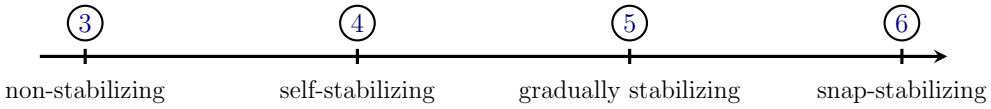


Figure 1.5 – Reading guide according to safety guarantees.

- **Anonymity (Figure 1.6):** We study different models of anonymity, from the (fully) anonymous model in Chapters 5 and 6, to the (fully) identified model in Chapter 4, by way of the in-between model of homonym processes in Chapter 3.

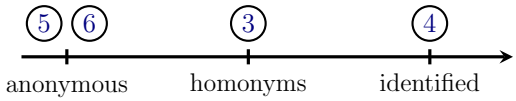


Figure 1.6 – Reading guide according to anonymity.

- **Considered Problem (Figure 1.7):** We can order contributions according to the considered problems. More precisely, we can differentiate whether the problem is *static*, *i.e.*, there is only one computation that ends after some finite time (*e.g.*, electing a leader), or *dynamic* (*e.g.*, token circulation), and whether the considered system is *dynamic*, *i.e.*, the topology may change over time, or *static*. Thus, in Chapters 3 and 4, we consider a static problem in static networks. In Chapter 6, we study a dynamic problem in static networks. Finally, in Chapter 5, we study a dynamic problem in dynamic networks.

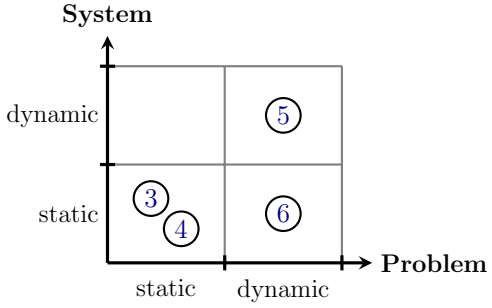


Figure 1.7 – Reading guide according to static or dynamic.

- **Considered Model (Figure 1.8):** Finally, we can differentiate the contributions according to the considered computational model, *i.e.*, locally shared memory model or message-passing model, and the considered daemon, *i.e.*, (distributed) weakly fair or (distributed) unfair. Thus, in Chapter 3, we consider the weakly fair daemon and the message-passing model. In Chapter 6, we also consider the weakly fair daemon but in the locally shared memory model. Finally, in Chapters 4 and 5, we consider the locally shared memory model under the unfair daemon.

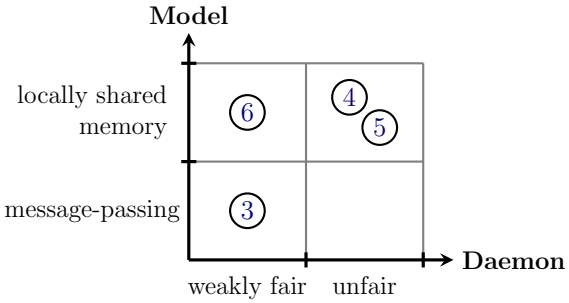


Figure 1.8 – Reading guide according to the model.

Chapter 2

Computational Model

"Problem-solving is hunting; it is savage pleasure and we are born to it."
— Thomas Harris, *The Silence of Lambs*

Contents

This chapter introduces the computational model used in this thesis. We first recall some notions from graph theory. Then, we present a model that generalizes every situation we will consider, namely communications through locally shared variables or messages, bidirectional or unidirectional networks, dynamic or static topology, anonymous or identified processes.

We present a general description of distributed systems (Section 2.2), distributed algorithms (Section 2.3), and their executions (Section 2.4).Then, this general model is instantiated depending on the proper characteristics of the models used in this thesis. We first present the message passing model in Section 2.5, the closest one to the implementation of a distributed system. In this model, processes exchange information by sending messages to each other. But, in this thesis, we mainly use a more abstract model, the locally shared memory model, presented in Section 2.6. This latter model focuses

on local state updates, *i.e.*, instead of receiving messages containing information on the state of a neighboring process q, a process p can directly read the state of q.

Finally, we formally define some fault-tolerant properties of distributed systems in Section 2.7.

For every notation introduced in this chapter, the subscript ALG referring to the considered algorithm can be omitted for sake of simplicity when ALG is clear from the context.

2.1 Preliminaries

In this section, we present some notions and notations from graph theory which are useful to describe the topology of a network. For every notation introduced in this section, the subscript $\mathcal{G}$ or $\mathcal{T}$ referring to the considered graph or tree can be omitted for sake of simplicity when $\mathcal{G}$ or $\mathcal{T}$ is clear from the context.

2.1.1 Graph

A *(simple) (finite) graph* $\mathcal{G} = (V, E)$ is a pair composed of a finite set V of *vertices* (or *nodes*) and a finite set $E \subseteq V \times V$ of ordered pairs of distinct vertices (*i.e.*, we exclude self-loops), called *edges*. We denote by n the number of vertices $|V|$.

$\mathcal{G}$ is *undirected* if, for every edge $(u, v) \in E$, $(v, u) \in E$. Otherwise, $\mathcal{G}$ is said to be *directed*. If $\mathcal{G}$ is undirected, we can denote by $\{u, v\}$ both (u, v) and (v, u). We say that v is a *successor* of u in $\mathcal{G}$ if $(u, v) \in E$. On the contrary, we say that v is a *predecessor* of u in $\mathcal{G}$ if $(v, u) \in E$. We denote by $\Gamma^+_{\mathcal{G}}(u)$ (respectively, $\Gamma^-_{\mathcal{G}}(u)$) the set of successors (respectively, predecessors) of u in $\mathcal{G}$. Let $\Gamma_{\mathcal{G}}(u) = \Gamma^+_{\mathcal{G}}(u) \cup \Gamma^-_{\mathcal{G}}(u)$. In an undirected graph, $\Gamma_{\mathcal{G}}(u) = \Gamma^+_{\mathcal{G}}(u) = \Gamma^-_{\mathcal{G}}(u)$ and vertices in $\Gamma_{\mathcal{G}}(u)$ are said to be the *neighbors* of u in $\mathcal{G}$.

A sequence $v_0, v_1, \ldots, v_k$ of vertices is a *path* from v_0 to v_k if $\forall i \in \{0, \ldots, k-1\}$, $(v_i, v_{i+1}) \in E$. The *length* of a path is the number k of edges it is made of. We say that v_0 (respectively v_k) is the *initial* (respectively, *terminal*) *extremity* of the path. A *simple path* is a path without any repeated edge. An *elementary path* is a path without any repeated vertex. A *cycle* is a path where the initial and the terminal extremities are the same vertex. It is often called *circuit* in directed graphs. A *simple circuit* is a circuit without repeated vertex, except the initial and terminal extremity, and without repeated edge. A *simple cycle* is a cycle without repeated vertex, except the initial and terminal extremity, and without repeated edge.

A graph $\mathcal{G}$ is *connected* if, for every pair of vertices u and v, there is a path from u to v in $\mathcal{G}$. Otherwise, we say that $\mathcal{G}$ is *disconnected*.

$\mathcal{G}' = (V', E')$ is a *subgraph* of $\mathcal{G} = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. Given $V' \subseteq V$, the *subgraph of* $\mathcal{G}$ *induced by* V' is (V', E'), where $E' = \{(u, v) \in E : u \in V' \wedge v \in V'\}$. A *connected component* $\mathcal{G}' = (V', E')$ of $\mathcal{G} = (V, E)$ is a maximal connected subgraph, *i.e.*, $\mathcal{G}'$ is connected and there is no edge in E between a vertex of V' and a vertex of $V \backslash V'$.

The *distance* from u to v in $\mathcal{G}$ is the length of a shortest path from u to v and is denoted

by $\|u, v\|_{\mathcal{G}}$. If there is no path from u to v, we conventionally define $\|u, v\|_{\mathcal{G}} = \infty$. The *diameter* $\mathcal{D}_{\mathcal{G}}$ of a graph $\mathcal{G}$ is the maximum distance between any two processes in $\mathcal{G}$. Notice that if a graph is disconnected, we conventionally say that its diameter is infinite.

In an undirected graph, the *degree* of a vertex u in $\mathcal{G}$ denoted $\delta_{\mathcal{G}}(u)$, is the number of neighbors, *i.e.*, $\delta_{\mathcal{G}}(u) = |\Gamma_{\mathcal{G}}(u)|$. We denote $\Delta_{\mathcal{G}} = \max\{\delta_{\mathcal{G}}(u) : u \in V\}$, the *degree* of $\mathcal{G}$. In a directed graph, we distinguish the *outdegree*, denoted by $\delta^+_{\mathcal{G}}(u)$, and the *indegree*, denoted by $\delta^-_{\mathcal{G}}(u)$, of vertex u. The outdegree of u is the number of successors of u, *i.e.*, $\delta^+_{\mathcal{G}}(u) = \left|\Gamma^+_{\mathcal{G}}(u)\right|$. The indegree of u is the number of predecessors of u, *i.e.*, $\delta^-_{\mathcal{G}}(u) = \left|\Gamma^-_{\mathcal{G}}(u)\right|$.

A graph $\mathcal{G} = (V, E)$ is *isomorphic* to another graph $\mathcal{G}' = (V', E')$ if and only if there exists a bijective function $f : V \to V'$ such that for any two processes $u, v \in V$, $(u, v) \in E \Leftrightarrow (f(u), f(v)) \in E'$.

2.1.2 Rings

A graph $\mathcal{R} = (V, E)$ is a *directed ring* if it is isomorphic to a simple circuit. Similarly, a graph $\mathcal{R} = (V, E)$ is an *(undirected) ring* if it is isomorphic to a simple cycle.

If the ring is directed, we say that the predecessor of a vertex u is its *left* neighbor while its successor is its *right* neighbor.

2.1.3 Trees and Forests

A graph $\mathcal{T} = (V, E)$ is a *tree* if it is a connected acyclic undirected graph. It is composed of $|V| - 1$ edges. A *forest* is an acyclic graph that may be disconnected, all its connected components are trees.

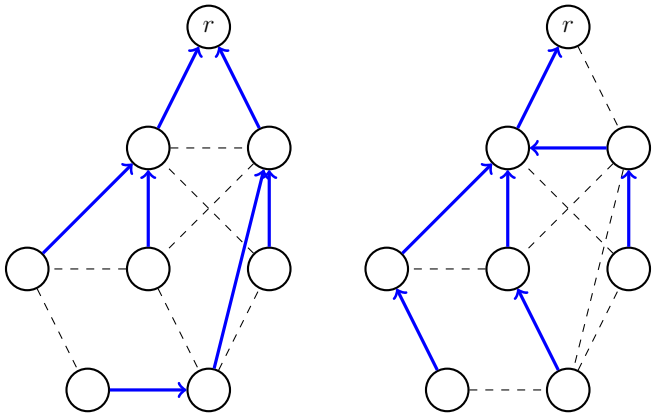
A *rooted* tree is a tree in which a vertex has been distinguished as the *root*. In a rooted tree, a vertex v is the *parent* of a vertex $u \neq r$ if v is the adjacent vertex of u on the shortest path from u to the root r. In this case, we also say that u is a *child* of v. A vertex without children is a *leaf*. By definition, the root has no parent. Conventionally, we denote by $\perp$ the parent of the root.

If the tree is rooted, it can be oriented, *i.e.*, we can orient the edges either away from the root, in this case it is called an *out-tree*, or towards the root, in this case it is called an *in-tree*. In this thesis, we will only consider in-trees.

The *level* of a vertex v in a tree $\mathcal{T} = (V, E)$ rooted at r is denoted $lvl_{\mathcal{T}}(v)$. $lvl_{\mathcal{T}}(v)$ is the distance from v to r, *i.e.*, $lvl_{\mathcal{T}}(v) = \|v, r\|_{\mathcal{T}}$. The *height* of a tree is the maximum level of its vertices.

An *ancestor* of u is any vertex v on the shortest path from u to r. On the contrary, a *descendant* of u is any vertex v such that u is on the shortest path from v to r. The *subtree* of v is the subgraph induced by v and its descendants.

$\mathcal{T} = (V', E')$ is a *spanning tree* of $\mathcal{G} = (V, E)$ if $\mathcal{T}$ is a tree such that $V' = V$ and $E' \subseteq E$. $\mathcal{T}$ is a *breadth-first search (BFS) spanning tree* of $\mathcal{G}$ if, for every vertex $v \in V$, the



(a) BFS spanning tree **(b)** Non BFS spanning tree

Figure 2.1 – Example of BFS spanning tree and of non BFS spanning tree rooted at some vertice r. Dashed edges do not belong to the tree.

distance from v to the root r through $\mathcal{T}$ is the exact distance in $\mathcal{G}$, *i.e.*, $lvl_{\mathcal{T}}(v) = \|v, r\|_{\mathcal{G}}$. See an example on Figure 2.1.

2.2 Distributed System

A *distributed system* is a set of autonomous but interconnected computational units, called *processes*. Autonomous means that there is no central control over the processes and they do not share a central memory. Interconnected means that they are able to exchange information.

Communications. Each process p is able to communicate with a subset of processes. More precisely, a process can get information from its *predecessors* and can give information to its *successors*. These communication capabilities may change over time. If such changes are assumed to be possible, the network is said to be *dynamic*. Otherwise, it is said to be *static*.

Furthermore, the communications can be *bidirectional*, *i.e.*, a process p can give information to another process q if and only if q can give information to p. Otherwise, we say that the communications are *unidirectional*. If the communications are bidirectional, every predecessor of a process is also one of its successor, and the other way round. Processors and successors are then called *neighbors*.

In the message passing model, those communications are carried out by sending information through channels. In the locally shared memory model, processes communicates using locally shared variables.

The topology of the system, and so the communication capabilities of the processes, at a given time can be modeled by a graph $\mathcal{G} = (V, E)$. V is the set of processes that are in the system. E is the set of communication links, *i.e.*, $p \in V$ can give information to $q \in V$ if and only if $(p, q) \in E$. Notice that $\mathcal{G}$ is undirected if and only if the communications

are bidirectional.

Process state. Processes are computational units with a (finite) local memory. They can store values into a finite number of *variables*. We denote by $p.x$ the variable x of process p.

Some of those variables can be *inputs*, *i.e.*, read-only variables whose value is set, and may be updated over time. If those variables are not constant, they can only be modified by the environment of the system (*e.g.*, the user or another algorithm) but not by the algorithm. We consider that the topology is an input. (Notice that its value may change if the network is dynamic.) We denote by $\mathcal{I}_{\text{ALG}}$ the set of input variables of algorithm ALG.

Variables can also be *outputs*, *i.e.*, variables used to return the result of the computation. Variables that are neither inputs nor outputs are called *internal* variables.

The *(local) state* of a process is then the vector of values of its variables.

Identities. A process distinguishes its neighbors using local labels. We denote by $p.\mathcal{N}^-$ and $p.\mathcal{N}^+$ the set of local labels of the predecessors and successors of p, respectively. If the network is bidirectional, $p.\mathcal{N} = p.\mathcal{N}^- = p.\mathcal{N}^+$. By abuse of notation, we denote by q the local label of process q at its neighbor p.

In addition to the local labeling, each process p may have a name or *identity (ID)*, *p.id*. We denote by $\mathfrak{id}$ the set of all possible IDs. We assume that the number of bits required to stock an ID is b. We assume that values of ID type can be compared (order and equality).

- If every process has a unique ID, the system is *(fully) identified*.
- If several processes have the same ID, there are *homonym* processes. In this case, IDs are called *labels*. For any label ℓ, let $\text{mlty}(\ell)$ be the *multiplicity* of ℓ in the network, *i.e.*, the number of processes whose label is ℓ.
- If every process has the same ID or if there is no ID at all, the processes are *(fully) anonymous*. It is impossible to distinguish two processes except maybe by their degree (*i.e.*, their number of neighbors). In particular, they have the same local algorithm.
- Processes can also be *semi-anonymous*, *i.e.*, only some processes are distinguished (by their role, their inputs, *etc.*), *e.g.*, the root in a rooted tree network may not have the same local algorithm than other processes. Notice that semi-anonymous processes is a particular case of homonym processes.

2.3 Distributed Algorithm

2.3.1 Algorithm

Each process p updates its state according to a *local algorithm* ALG_p. A *distributed algorithm* ALG is the collection of all local algorithms. Each local algorithm is written as

a set of *guarded actions* of the following form:

$$\langle \text{label} \rangle :: \langle \text{guard} \rangle \rightarrow \langle \text{statement} \rangle$$

The *labels* are used to identify the actions in the reasoning. The *guard* of an action is a Boolean expression. Informally, in the message passing model, the guard involves the state of the process (*i.e.*, the values of its variables) and on the messages received by the process. Similarly, in the locally shared memory model, the guard involves the state of the process and the variables shared with its neighbors. Further details will be given when the corresponding models will be instantiated. The *statement* updates the state of the process, *i.e.*, its (writable) variables. Moreover, in the message passing model, statements may contain message sending.

When the guard of an action is evaluated to TRUE, the action is said to be *enabled*. If at least one action is enabled at a process p, p is also said to be *enabled*. An action can be executed only if it is enabled. In this case, the execution of the action consists in executing its statement. The evaluation of the guards and the execution of the statement are assumed to be *atomic*.

Priorities. Some algorithms are designed using *priorities* to simplify the guards of actions. In this case, actions have the following form:

$$\langle \text{label} \rangle \quad (\textit{prio. } \langle \text{priority} \rangle) :: \langle \text{guard} \rangle \rightarrow \langle \text{statement} \rangle$$

An action is enabled at a process p if its guard is evaluated to TRUE at p and no higher priority action is also enabled. An action of priority i is said to be of *higher priority* (respectively, *lower priority*) than any action with priority $j > i$ (respectively, $j < i$).

We can rewrite the local algorithm as an equivalent one without priorities. Consider a local algorithm of $k \geq 1$ actions "L_i (*prio.* P_i) :: $G_i \rightarrow S_i$", $i \in \{1, \ldots, k\}$. Let $HP(L_i)$ be the subscripts of actions with a higher priority than L_i-action. We denote $L'_i :: G'_i \rightarrow S_i$, $i \in \{1, \ldots, k\}$, the actions of the resulting local algorithm without priorities, where:

$$G'_i \equiv G_i \wedge \bigwedge_{j \in HP(L_i)} \neg G_j$$

Notice that the guard of the highest priority actions does not change.

2.3.2 Configuration

For a given distributed system and a given algorithm ALG, we denote by $\mathcal{S}_{\text{ALG}}$ the set of all possible local states. We also denote by $\mathcal{L}_{\text{ALG}}$ the set of all possible states for a communication link. A *configuration* γ_i of the system under the algorithm ALG is a tuple $\gamma_i = (\mathcal{G}_i, V_i \rightarrow \mathcal{S}_{\text{ALG}}, E_i \rightarrow \mathcal{L}_{\text{ALG}})$, where:

$\mathcal{G}_i = (V_i, E_i)$ is a graph which models the topology of the network in configuration γ_i.

$V_i \rightarrow \mathcal{S}_{\text{ALG}}$ is a function which associates a state to any process of V_i. We denote by $\gamma_i(p) \in \mathcal{S}_{\text{ALG}}$ the state of $p \in V_i$ in configuration γ_i. We denote $\gamma(p).x$ the value of variable $p.x$ in configuration γ.

$E_i \to \mathcal{L}_{\text{ALG}}$ is a function which associates a state to any communication link of E_i. This parameter is relevant only if the considered model contains channels. We denote by $\gamma_i(L_{(p,q)}) \in \mathcal{L}_{\text{ALG}}$ the state of the link (p, q) in configuration γ_i. Notice that, in the locally shared memory model, this function is not relevant, and so is not considered in the configuration.

We denote by $\mathcal{C}_{\text{ALG}}$ the set of every possible configurations of the system under algorithm ALG.

2.4 Execution of Distributed Algorithms

Informally, an execution of a distributed algorithm ALG is a sequence of configurations $e = (\gamma_i)_{i\geq 0}$. A pair of two successive configurations in e is called a step. During a step, an adversary, the *daemon*, triggers the activation of some processes and/or the modification of some input values.

2.4.1 Step

A *step* of algorithm ALG is a pair of configurations γ_i and γ_{i+1} such that we can reach γ_{i+1} from γ_i by

- activating some processes

  and/or

- performing input value changes, in particular topological changes.

Activation of processes and inputs updates are all performed atomically. The set of all possible steps induces a binary relation over configurations, denoted $\mapsto_{\text{ALG}} \subseteq \mathcal{C}_{\text{ALG}} \times \mathcal{C}_{\text{ALG}}$.We denote by $\mapsto^{+}_{\text{ALG}}$ the transitive relation generated by $\mapsto_{\text{ALG}}$.

Process Activation. A process can be activated during $\gamma_i \mapsto_{\text{ALG}} \gamma_{i+1}$ only if it is enabled in γ_i. In a step, activated processes execute one of their enabled actions in ALG.

If the input values also change between γ_i and γ_{i+1}, the execution of actions during $\gamma_i \mapsto_{\text{ALG}} \gamma_{i+1}$ depends on the input values in γ_i.

Topological Changes. If the topology of the system changes between γ_i and γ_{i+1}, *i.e.*, $\mathcal{G}_i \neq \mathcal{G}_{i+1}$, then the step $\gamma_i \mapsto_{\text{ALG}} \gamma_{i+1}$ contains a finite (yet unbounded) number of topological changes of the following kinds.

A process p can *join* the system, *i.e.*, $p \notin V_i$ but $p \in V_{i+1}$. This event, denoted by join_p, triggers the atomic execution of a particular action called *bootstrap*. The bootstrap action initializes the state of p to a particular state, called *bootstate*, meaning that the output of p is meaningless for now. This action is executed instantly, without any communication. We denote by $New(k)$ the set of processes that are in bootstate in γ_k. More precisely, when p joins the system in $\gamma_i \mapsto \gamma_{i+1}$, we have $p \in New(i+1)$, but $p \notin New(i)$. Moreover, until p executes its very first action, say in step $\gamma_x \mapsto \gamma_{x+1}$, it is still in bootstate, *i.e.*, $\forall k \in \{i+1, \ldots, x\}$, $p \in New(k)$, but $p \notin New(x+1)$.

A process p can also *leave* the system, *i.e.*, $p \in V_i$ but $p \notin V_{i+1}$. Every communication link from or to p is also deleted.

Finally, some communication link can *appear* or *disappear* between two different processes p and q, *i.e.*, $(p,q) \notin E_i \wedge (p,q) \in E_{i+1}$ or $(p,q) \in E_i \wedge (p,q) \notin E_{i+1}$, respectively.

Classification of Steps. We distinguish different types of steps.

We call *dynamic step* a step containing at least one topological change. On the contrary, we call *static step* a step containing no topological change. We denote by $\mapsto^{d}_{\text{ALG}}$ the relation defining all dynamic steps and by $\mapsto^{s}_{\text{ALG}}$ the relation defining all static steps.

We call *activation step* a step containing at least one process activation.

The set of steps is partitioned into dynamic and static steps. However, an activation step can be either a dynamic step, if it contains at least one topological change, or a static step, otherwise.

We can also differentiate dynamic steps. In particular, we might make assumptions on allowed dynamic steps, *i.e.*, restrict the set of possible dynamic steps w.r.t. the possible topological changes. To that goal, we define a binary predicate ρ over graphs, called *dynamic pattern*. Let $\mapsto^{d,\rho}_{\text{ALG}} = \left\{(\gamma_i, \gamma_{i+1}) \in \mathcal{C}^2_{\text{ALG}} : \gamma_i \mapsto^{d}_{\text{ALG}} \gamma_{i+1} \wedge \rho(\mathcal{G}_i, \mathcal{G}_{i+1})\right\}$ be the subrelation of $\mapsto^{d}_{\text{ALG}}$ induced by ρ. Every step in $\mapsto^{d,\rho}_{\text{ALG}}$ is called a *ρ-dynamic step*.

2.4.2 Daemon

The asynchronism (whether and when an enabled process is activated) and environment (whether and when an input value is modified) of the system is modeled by an adversary, called *daemon*. We say that an execution $e = (\gamma_i)_{i\geq 0}$ is *driven* by a daemon $\mathfrak{D}$ if e satisfies the hypotheses $\mathfrak{D}$ on the asynchronism and the environment of the system.

Locality Constraints. The daemon can be constrained on the locality of the input value modifications and processes activation. We focus here on the locality constraints on process activation without any constraint on input value modifications, but the following definitions can easily be extended.

Let consider an execution $e = (\gamma_i)_{i\geq 0}$ of algorithm ALG. For every $i \geq 0$, we denote by $Act_i(e)$ the set of processes that are activated during step $\gamma_i \mapsto_{\text{ALG}} \gamma_{i+1}$ of e.

From a general point of view, a daemon is said to be *k-central* [DT11] if it cannot activate two enabled processes whose distance is lower than k. More formally, $\forall e = (\gamma_i)_{i\geq 0}$, $\forall i \geq 0$,

$$(p \in Act_i(e) \wedge q \in Act_i(e) \wedge p \neq q) \Leftrightarrow \left(\|p,q\|_{\mathcal{G}_i} > k \wedge \|q,p\|_{\mathcal{G}_i} > k\right)$$

Three different locality constraints are mainly used in the literature:

- The 0-central or *distributed* daemon, *i.e.*, the more general one where the daemon has no locality constraint.

- The 1-central or *locally central* daemon, *i.e.*, two neighbors cannot be activated during the same step.
- The $\mathcal{D}_{\mathcal{G}_i}$-central, *central*, or *sequential* daemon, *i.e.*, the more constrained one where only one process can be activated at each step $\gamma_i \mapsto_{\text{ALG}} \gamma_{i+1}$.

Fairness Constraints. We can also constrained the daemon on its fairness, *i.e.*, when the daemon should activate a process or trigger an input value modification. It models the speed of the different processes and frequency of the environment change.

The weakest assumption on fairness is the *unfair* daemon: no fairness constraint is assumed. The unfair daemon was initially defined for networks where input values (in particular, the topology) are constant over the execution. In those networks and under an unfair daemon, an enabled process may never be activated unless it is the only enabled one.

Nonetheless, in a context where the input values may change over time, we must extend this definition: the unfairness does not only rely on process activation but also on input value changes. Hence, we can have executions with an infinite number of input value changes, and, in particular, an infinite number of dynamic steps. Under these assumptions, it is trivially impossible to solve any (non-trivial) problem. So, there are two possibilities: either assuming some fairness constraints on input value changes, see for example [CFQS12] for fairness constraints on dynamic steps, or bounding their number. Now, even if we assume some fairness constraints on input value changes, it remains impossible to solve some problems. For example, in [BDKP16], Braud-Santoni et al. showed that it is impossible to deterministically solve the building of some classic overlay structures in dynamic networks where there exists infinitely often a path between any two processes. Indeed, without any knowledge on the frequency of dynamic steps, the system cannot converge and maintain a correct structure. So, in the following, we always assume that the number of input value changes (in particular, the number of dynamic steps) is bounded or we use some constraints on these changes.

Two other fairness assumptions are also often considered. If the daemon is *(weakly) fair*, a process that is continuously enabled along an execution is eventually activated. Finally, if the daemon is *strongly fair*, a process that is enabled infinitely often is activated infinitely often.

Other Constraints. The *synchronous* daemon is constrained both on the locality and the fairness since it selects every enabled process at each step.

2.4.3 Execution

An *execution of* ALG is a sequence of configurations $e = (\gamma_i)_{i\geq 0}$ such that $\forall i \geq 0$, $\gamma_i \mapsto_{\text{ALG}} \gamma_{i+1}$.

We denote by $\mathcal{E}^{\tau}_{\text{ALG}}$ the set of maximal executions of ALG which contains at most τ dynamic steps. Any execution $e \in \mathcal{E}^{\tau}_{\text{ALG}}$ is either infinite, or ends in a so-called *terminal* configuration, where all processes in the system are disabled. The set of all possible

maximal executions is therefore equal to $\cup_{\tau \geq 0} \mathcal{E}^{\tau}_{\text{ALG}}$. When clear from the context, $\mathcal{E}^{0}_{\text{ALG}}$ is simply denoted by $\mathcal{E}_{\text{ALG}}$.

For any subset of configuration $X \subseteq \mathcal{C}_{\text{ALG}}$, we denote by $\mathcal{E}_{\text{ALG}}(X)$ (respectively, $\mathcal{E}^{\tau}_{\text{ALG}}(X)$) the set of maximal executions in $\mathcal{E}_{\text{ALG}}$ (respectively, $\mathcal{E}^{\tau}_{\text{ALG}}$) that start from a configuration of X, *i.e.*,

$$\begin{aligned} \mathcal{E}_{\text{ALG}}(X) &= \{(\gamma_i)_{i\geq 0} \in \mathcal{E}_{\text{ALG}} : \gamma_0 \in X\} \\ \mathcal{E}^{\tau}_{\text{ALG}}(X) &= \{(\gamma_i)_{i\geq 0} \in \mathcal{E}^{\tau}_{\text{ALG}} : \gamma_0 \in X\} \end{aligned}$$

For sake of simplicity, we denote by $\mathcal{E}^{\tau,\rho}_{\text{ALG}}$ the set of maximal executions of ALG that contains at most τ ρ-dynamic steps (and no other dynamic steps).

Static Network. If the algorithm ALG is designed to be executed on a static network, *i.e.*, the topology of the system remains the same during the whole execution ($\mathcal{G}_i$ is equal to $\mathcal{G}_0$ for every $i \geq 1$).

In this latter case, we can simplify notations. We denote by:

- $\mathcal{G} = (V, E) = \mathcal{G}_0$, the graph modeling the topology of the system,
- $n = |V|$, the number of processes,
- $\mathcal{D}$, the diameter of the network, and
- $\mathcal{E}_{\text{ALG}}$, the set of all possible executions.

2.5 Message Passing Model

In the *(asynchronous) message passing* model [Lyn96, Tel00], processes communicate by sending messages through communication links. The state of a communication link (p, q), denoted by $L_{(p,q)}$, is the ordered list of messages it contains. We assume FIFO links, thus this order satisfies the partial order of insertion into the link, *i.e.*, the messages already in the link in the initial configuration are the first ones but their order is arbitrary, the other messages are sorted by their order of appearance.

$p.\mathcal{N}^{-}$ may not be known by the process, until receiving messages from its predecessors. $p.\mathcal{N}^{+}$ (or $p.\mathcal{N}$ in a bidirectional network) is an input and its value may change if the network is dynamic. Hence, a process p is aware of the local topology and of topological changes. In particular, if the communication link between two processes p and q disappears during step $\gamma_i \mapsto_{\text{ALG}} \gamma_{i+1}$, the messages contained in the channel are lost.

The processes handle messages using functions **send** and **rcv**. In this thesis, we assume that the links are reliable, *i.e.*, no message is lost (except in case of topological changes), so calls to $\mathbf{send}_q$ by p and $\mathbf{rcv}_p$ by q are the only way to modify $L_{(p,q)}$. Since the links are assumed to be FIFO, when p executes $\mathbf{send}_q$ m, the message m is added at the tail of $L_{(p,q)}$, and the head of $L_{(p,q)}$ will be the first message processed by q on this channel.

More precisely, each message has the following form $\langle x_1, \ldots, x_k \rangle$, where $x_1, \ldots, x_k$ is a list of values, each of them of a given data type. We say that x *conforms* to y if y is a

value and $x = y$, or if y is a variable and has the same data type than x. By extension, we say that a message $m = \langle x_1, \ldots, x_k \rangle$ *conforms* to pattern $\langle y_1, \ldots, y_{k'} \rangle$ if and only if $k = k'$ and $\forall i \in \{1, \ldots k\}$, x_i conforms to y_i.

Then, no message is lost, so a message remains in $L_{(p,q)}$ until q (explicitly) receives it by a call to **rcv**. A call to $\mathbf{rcv}_s \langle y_1, \ldots, y_{k'} \rangle$ considers the head message $\langle x_1, \ldots, x_k \rangle$ of an arbitrary incoming channel $L_{(p,q)}$ of q. This call returns TRUE if and only if:

- $s \in p.\mathcal{N}^-$
- $\langle x_1, \ldots, x_k \rangle$ conforms to $\langle y_1, \ldots, y_{k'} \rangle$.

In a guarded action $\langle \text{label} \rangle :: \langle \text{guard} \rangle \rightarrow \langle \text{statement} \rangle$ of a process p, the guard holds on the variables of p and on calls to **rcv**. The statement modifies the values of the variables of p and/or calls to **send**. If the guard contains a call to $\mathbf{rcv}_s \langle y_1, \ldots, y_k \rangle$, there are three cases:

- If the guard is evaluated to FALSE, even if the call to $\mathbf{rcv}_s$ returns TRUE, the call to $\mathbf{rcv}_s$ does not modify the state of the incoming links.
- If the guard is evaluated to TRUE (in particular, the call to $\mathbf{rcv}_s$ returns TRUE) but p is not activated or does not execute this action, the call to $\mathbf{rcv}_s$ does not modify the state of the incoming link $L_{(s,p)}$.
- If the guard is evaluated to TRUE (in particular, the call to $\mathbf{rcv}_s$ returns TRUE) and p executes this action, the considered head message $\langle x_1, \ldots, x_k \rangle$ sent by some process q is removed from the corresponding channel $L_{(q,p)}$ (a message cannot be received several times), the value p is assigned to s if s is a variable, and, $\forall i \in \{1, \ldots, k\}$, the value x_i is assigned to y_i if y_i is a variable.

Time Complexity Unit. In the message passing model, we evaluate time complexity in terms of *time units* [Tel00], where the message transmission lasts at most one time unit and the process execution time is zero. Roughly speaking, the time unit is a measure according to the slowest messages. Indeed, the execution is normalized such that the longest message delay (*i.e.*, the transmission of the message followed by its processing at the receiving process) becomes one time unit.

2.6 Locally Shared Memory Model

The *locally shared memory model* was first introduced by Dijkstra in [Dij74]. In this model, processes communicate through a finite set of *locally shared variables*. A process can read its variables and the ones of its predecessors, but it can only modify the value of its own variables. $p.\mathcal{N}^-$ (or $p.\mathcal{N}$ in a bidirectional network) is an input whose value may change if the network is dynamic. Similarly to the message passing model, a process is aware of its local topology and of topological changes.

Since there is no message channels, a configuration γ_i is the tuple $\gamma_i = (\mathcal{G}_i, V_i \rightarrow \mathcal{S}_{\text{ALG}}, \perp)$. Hence, we simply denote by $\gamma_i = (\mathcal{G}_i, V_i \rightarrow \mathcal{S}_{\text{ALG}})$. The state $\gamma_i(p)$ of process p is the vector of values of all the variables of p. By abuse of notation, we denote by $\gamma_i(p).x$ the value of variable $p.x$ in configuration γ_i.

In a guarded action $\langle \text{label} \rangle :: \langle \text{guard} \rangle \rightarrow \langle \text{statement} \rangle$ of a process p, the guard holds on the variables of p and its neighbors. The statement modifies the variables of p.

Time Complexity Units. In this model, we mainly use two different time units to measure the time complexity of a distributed algorithm. The first one is simply the number of (activation) steps.

We can also measure the time complexity in terms of *(asynchronous) rounds*. The asynchronous rounds were first introduced by Dolev et al. in [DIM97], but we use here the extended definition of Bui et al. in [BDPV07]. Roughly speaking, the round is a measure according to the speed of the slowest process.

We first need to define the *neutralization* of a process p during step $\gamma_i \mapsto_{\text{ALG}} \gamma_{i+1}$. p is neutralized in $\gamma_i \mapsto_{\text{ALG}} \gamma_{i+1}$ if it is enabled in γ_i but either p is no more in the system in the next configuration γ_{i+1} (in a dynamic network), or p is no more enabled in γ_{i+1} but does not execute an action during step $\gamma_i \mapsto_{\text{ALG}} \gamma_{i+1}$.

The first round of an execution $e = (\gamma_i)_{i \geq 0}$ is the minimal prefix e' of e where every enabled process in configuration γ_0 either executes an action or is neutralized. Let γ_j, $j \geq 0$, be the last configuration of e'. The second round of e is the first round of $e' = (\gamma_i)_{i \geq j}$, and so on.

2.7 Self-stabilization and Snap-stabilization

In this section, we define fault-tolerant properties for distributed systems. Notice that self-stabilization and snap-stabilization were defined for static networks. Hence, except stated otherwise, we consider in this section only static networks.

We first define notions classically used in self-stabilization. A specification SP is a predicate over sequences of configurations. Let ALG be a distributed algorithm, SP be a specification, and $X, Y \subseteq \mathcal{C}_{\text{ALG}}$ be two subsets of configurations.

- X is *closed* under ALG if and only if $\forall \gamma \in X, \forall \gamma' \in \mathcal{C}_{\text{ALG}}$, $\gamma \mapsto_{\text{ALG}} \gamma \Rightarrow \gamma' \in X$.
- Y *converges* to X under ALG if and only if

$$\forall e = (\gamma_i)_{i \geq 0} \in \mathcal{E}^0_{\text{ALG}}(Y), \exists i \geq 0, \gamma_i \in X$$

- ALG *stabilizes* from Y to specification SP by X if and only if:
  1. *Closure:* X is closed under ALG.
  2. *Convergence:* Y converges to X under ALG.
  3. *Correctness:* $\forall e \in \mathcal{E}^0_{\text{ALG}}(X), SP(e)$.
- The *convergence time* from Y to X is the maximal time (in terms of time units, steps, or rounds) to reach a configuration of X in every execution of $\mathcal{E}^0_{\text{ALG}}(Y)$.

Self-stabilization [Dij74]. Informally, an algorithm is self-stabilizing if, starting from an arbitrary configuration, the system converges in finite time to a legitimate configuration from which the specification is satisfied.

An algorithm ALG is *self-stabilizing* w.r.t. SP, if and only if there exists a non-empty subset of configurations $\mathscr{L} \subseteq \mathcal{C}_{\text{ALG}}$ such that ALG stabilizes from $\mathcal{C}_{\text{ALG}}$ (the set of every possible configuration) to SP by $\mathscr{L}$.

A configuration of $\mathscr{L}$ is called *legitimate*. Otherwise, we say that it is an *illegitimate* configuration. The *stabilization time* of ALG is the convergence time from $\mathcal{C}_{\text{ALG}}$ to $\mathscr{L}$.

Snap-stabilization [BDPV07]. Snap-stabilization is a variant of self-stabilization ensuring stronger safety guarantees. Indeed, an algorithm ALG is *snap-stabilizing* w.r.t. SP if and only if the specification is satisfied in any execution of ALG, *i.e.*, $\forall e \in \mathcal{E}^0_{\text{ALG}}$, $SP(e)$.

Chapter 3

Leader Election in Unidirectional Rings with Homonym Processes

"One Ring to rule them all, One Ring to find them, One Ring to bring them all and in the darkness bind them."

— J.R.R. Tolkien, *The Fellowship of the Ring*

Contents

3.1 Introduction

In this chapter, we consider the *leader election* problem, *i.e.*, we want to distinguish a unique process as the *leader*. More precisely, we consider the *(public) leader election* problem, *i.e.*, every process should eventually know the leader ID.

This problem is fundamental in distributed computing and has been extensively studied. Hence, we give here only the main results.

In 1980, Angluin [Ang80] showed the impossibility of solving deterministic leader election in networks of anonymous processes. Notice that this result still holds in the more restrictive bidirectional ring networks [Lyn96] and can trivially be extended to unidirectional ring networks.

From this negative result, two main lines of research have been considered: designing randomized solutions to leader election in anonymous networks, *e.g.*, [XS06, KPP+13], or deterministic solutions to leader election in identified networks, *e.g.*, [Lan77, CR79, Pet82].

Recently, the model of homonym processes has been introduced as an intermediate model between the anonymous networks and the identified networks. As stated in Section 2.2, homonym processes have IDs, called here *labels*, that may be not unique. In this chapter, we focus on the deterministic leader election in static unidirectional ring networks with homonym processes.

3.1.1 Related Work

Several recent works [YK89, FKK+04, DP04, DFT14, DP16] studied the leader election problem in networks with homonym processes.

Yamashita and Kameda study in [YK89] the feasibility of leader election in networks of arbitrary topology containing homonym processes. They propose a *process-terminating* (*i.e.*, every process eventually halts) leader election assuming that processes knows the size of the network.

In [FKK+04], Flocchini et al. study the *weak leader election* problem in bidirectional ring networks of homonym processes. This problem consists in distinguishing at least one process, if possible, and at most two processes. In this latter case, the two elected processes must be neighbors. Under the assumption that processes *a priori* know the number of processes, n, they show that the process-terminating weak leader election is possible if and only if the labeling of the ring is asymmetric, *i.e.*, there is no non-trivial rotational symmetry (non multiple of n) of the labels resulting in the same labeling. They also propose two process-terminating weak leader election algorithms for asymmetric labeled rings of n processes, assuming that n is prime and that there is only two different labels, 0 and 1. The first algorithm assumes a common sense of direction, *i.e.*, every process is able to distinguish its clockwise neighbor and its anti-clockwise neighbor. The second algorithm is a generalization of the first one, where the common sens of direction

is removed. No time complexity is given.

In [DFT14], Delporte et al. consider the leader election problem in bidirectional ring networks of homonym processes. They propose a necessary and sufficient condition on the number of distinct labels needed to solve the leader election problem. More precisely, they prove that there exists a solution to *message-terminating* (*i.e.*, processes do not halt but only a finite number of processes are exchanged) leader election problem in bidirectional rings if and only if the number of labels is strictly greater than the greatest proper divisor of n. Assuming this latter condition, they give two algorithms. The first one is message-terminating and does not assume any extra knowledge. On the contrary, the second algorithm is process-terminating but assumes the processes know n. They show that their second algorithm is asymptotically optimal in messages ($O(n \log n)$).

In [DP04], Dobrev and Pelc study a generalization of the process-terminating leader election problem in both unidirectional and bidirectional networks of homonym processes. They assume that processes *a priori* know a lower bound m and an upper bound M on the (unknown) number of processes, n. They propose algorithms that decide whether the election is possible and perform it, if so. They propose two synchronous algorithms, one for bidirectional and one for unidirectional rings, that both works in time $O(M)$ using $O(n \log n)$ messages. They also propose an asynchronous algorithm for bidirectional rings using $O(nM)$ messages and they show that it is optimal. No time complexity is given.

Similarly, in [DP16], Dereniowski and Pelc study a generalization of the process-terminating leader election problem in arbitrary networks of homonym processes where processes *a priori* know an upper bound k on the multiplicity of a given label ℓ that exists in the network. Precisely, each process knows that ℓ is the label of at least one but at most k processes. They propose a synchronous algorithm that, under these hypotheses, decide whether the election is possible and achieve it, if so. They show that this algorithm is asymptotically optimal in time ($O(k\mathcal{D} + \mathcal{D} \log(n/\mathcal{D}))$), where $\mathcal{D}$ is the diameter of the network. No space complexity is given.

3.1.2 Contributions

In this chapter, we study the leader election problem in static unidirectional ring networks with homonym processes under the message-passing model where, contrary to [FKK+04, DP04, DFT14], processes know neither the number of processes, n, nor any bounds on n.

We first show that the message-terminating leader election remains impossible to solve without any extra hypothesis (Section 3.3.1). Indeed, it is impossible to distinguish two processes with the same label in a symmetric labeled ring during a synchronous execution. So, we consider the class $\mathcal{A}$ of unidirectional ring networks with an asymmetric labeling.

We then show that the process-terminating leader election is impossible to solve in a unidirectional asymmetric ring where at least one label is unique (Section 3.3.3). We denote by $\mathcal{U}^*$ this subclass of ring networks. Notice that $\mathcal{U}^* \subseteq \mathcal{A}$.

Hence, we assume an additional knowledge. We assume that processes know an upper bound k on the multiplicity of labels. We denote $\mathcal{K}_k$ the class of unidirectional

rings containing no more than k processes with the same label. Under this hypothesis, the process-terminating leader election becomes possible in asymmetric rings. More precisely, it is possible to design process-terminating leader election algorithm for $\mathcal{A} \cap \mathcal{K}_k$. Notice that we also show the impossibility of message-terminating leader election in $\mathcal{K}_k$ (Section 3.3.1).

Then, we show that the message-terminating leader election in class $\mathcal{U}^* \cap \mathcal{K}_k$, $k \geq 2$, requires the exchange of at least $\Omega(kn + n^2)$ bits in the worst case. This lower bound is also valid for the super class $\mathcal{A} \cap \mathcal{K}_k$, $k \geq 2$.

In addition to the impossibility results, we propose three process-terminating leader election algorithms.

The first algorithm U_k (Section 3.4) solves the process-terminating leader election algorithm in $\mathcal{U}^* \cap \mathcal{K}_k$. Its time complexity is at most $n(k + 2)$ time units, its message complexity is $O(n^2 + kn)$, and it requires $\lceil \log(k + 1) \rceil + 2b + 4$ bits per process, where b is the number of bits required to store a label.

Furthermore, we show a lower bound of $\Omega(kn)$ time units on the time complexity of process-terminating leader election algorithms in $\mathcal{U}^* \cap \mathcal{K}_k$. Hence, U_k is asymptotically optimal. Notice that this lower bound is also valid for the upper class $\mathcal{A} \cap \mathcal{K}_k$.

Then, we propose two process-terminating leader election algorithms, A_k (Section 3.5) and B_k (Section 3.6), for the more general class $\mathcal{A} \cap \mathcal{K}_k$. Those two algorithms achieve the classical trade-off between time and space. A_k is asymptotically optimal in time, with at most $(2k + 2)n$ time units, but it requires $2(k + 1)nb + 2b + 3$ bits per process and at most $n^2(2k + 1)$ messages are exchanged during an execution. On the contrary, B_k requires only $2 \lceil \log k \rceil + 3b + 5$ bits per process (it is asymptotically optimal), but its time complexity is $O(k^2n^2)$ and its message complexity is $O(k^2n^2)$.

The impossibility results of Section 3.3.1 and U_k (Section 3.4) are published in the proceedings of the 18th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2016) [ADD+16a]. The lower bounds on time complexity (Section 3.3.2), the impossibility results of Section 3.3.3, A_k (Section 3.5), and B_k (Section 3.6), appears in the proceedings of the 31st International Parallel and Distributed Processing Symposium (IPDPS 2017) [ADD+17a]. A summary of these results is published in the proceedings of the 19èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications (ALGOTEL 2017) [Dur17].

3.2 Preliminaries

In this section, we detail the context, we introduce the considered specifications of leader election problem, and we formally define the three aforementioned ring classes.

3.2.1 Context

We consider static unidirectional ring networks with homonym processes in the message-passing model described in Section 2.5. We denote the $n \geq 2$ processes $p_0, \ldots, p_{n-1}$. A process p_i can only receive messages from its *left* neighbor, p_{i-1}, and can only send

messages to its *right* neighbor, p_{i+1}. Subscripts are modulo n. Hence, in this chapter, we simply denote **send** and **rcv** the function for sending and receiving messages, respectively.

In this chapter, we are not in a self-stabilizing context, hence we consider only executions that start in a particular *initial configuration*, where each process is in a designed *initial state* and every communication link is empty.

We consider executions driven by a distributed weakly fair daemon.

3.2.2 Leader Election

We consider two classic definitions of the problem of leader election in the message-passing model: the message-terminating and the process-terminating leader election. Informally, in a process-terminating solution, every process eventually halts, whereas, in a message-terminating solution, processes do not halt but only a finite number of messages is exchanged.

Definition 3.1 (*Message-terminating Leader Election*)

An algorithm ALG solves the *message-terminating leader election* problem in a ring network $\mathcal{R}$ if every execution e of ALG on $\mathcal{R}$ satisfies the following conditions:

1. e is finite.
2. Each process p has a Boolean variable $p.isLeader$ such that, in the terminal configuration of e, $L.isLeader$ is TRUE for a unique process L (*i.e.*, the leader).
3. Every process p has a variable $p.leader$ such that, in the terminal configuration, $p.leader = L.id$, where L satisfies $L.isLeader$.

Definition 3.2 (*Process-terminating Leader Election*)

An algorithm ALG solves the *process-terminating leader election* problem in a ring network $\mathcal{R}$ if it solves the message-terminating leader election in $\mathcal{R}$ and if every execution e of ALG on $\mathcal{R}$ satisfies the following additional conditions:

4. For every process p, $p.isLeader$ is initially FALSE and never switched from TRUE to FALSE: each decision of being the leader is irrevocable. Consequently, there should be at most one leader in each configuration.
5. Every process p has a Boolean variable $p.done$, initially FALSE, such that $p.done$ is eventually TRUE for all p, indicating that p knows that the leader has been elected. More precisely, once $p.done$ becomes TRUE, it will never become FALSE again, $L.isLeader$ is equal to TRUE for a unique process L, and $p.leader$ is permanently set to $L.id$.
6. Every process p eventually *halts*, *i.e.*, locally decides its termination, after $p.done$ becomes TRUE.

3.2.3 Ring Networks Classes

An algorithm ALG solves the message-terminating (resp. process-terminating) leader election *for the class of ring networks* C if it solves the message-terminating (resp. process-terminating) leader election for every ring network $\mathcal{R} \in C$. In particular, ALG cannot be given any specific information about the network (such as its cardinality or the actual multiplicity of labels) unless that information holds for all ring networks of C. Indeed, ALG must work for every $\mathcal{R} \in C$ without any change whatsoever in its code.

A ring network $\mathcal{R}$ of n processes is said to be *symmetric* if a non-trivial rotation of the labels results in the same labeling, *i.e.*, there is some integer $0 < d < n$ such that, for all $i \geq 0$, p_i and p_{i+d} have the same label. Otherwise, $\mathcal{R}$ is said to be *asymmetric*.

We consider three classes of ring networks:

- $\mathcal{A}$ is the class of all asymmetric ring networks.
- $\mathcal{U}^*$ is the class of all ring networks in which at least one process has a unique label. By definition, $\mathcal{U}^* \subseteq \mathcal{A}$.
- $\mathcal{K}_k$, with $k \geq 1$ a given integer, is the class of all ring networks where no more than k processes have the same label. Notice that k is an upper bound on the multiplicity of labels in $\mathcal{R} \in \mathcal{K}_k$.

3.3 Impossibility Results and Lower Bounds

In this section, we present our impossibility results and lower bounds on the time complexity and the amount of exchanged information.

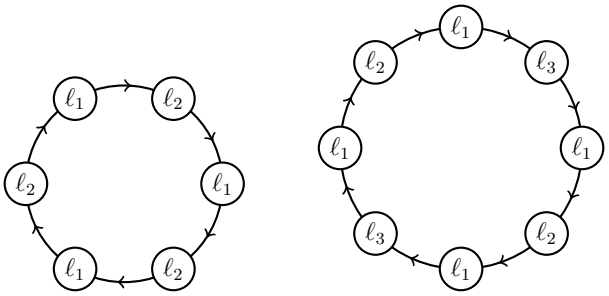
3.3.1 Symmetric Rings and Class $\mathcal{K}_k$

Theorem 3.1

There is no algorithm that solves message-terminating leader election in symmetric rings.

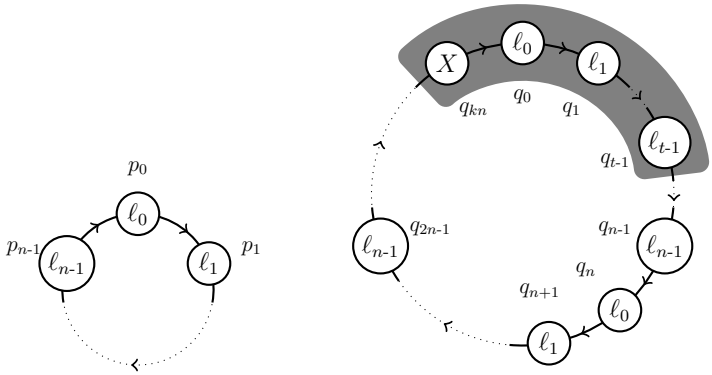
Proof: Let $\mathcal{R}$ be a symmetric ring of $n \geq 2$ processes. Let $0 < d < n$ such that, for all $i \geq 0$, p_i and p_{i+d} have the same label. Assume by contradiction that ALG is a message-terminating leader election algorithm for $\mathcal{R}$. Let $e = (\gamma_j)_{j \geq 0}$ be the synchronous execution of ALG on $\mathcal{R}$. At every step of e, each p_i, $i \geq 0$, makes exactly the same actions as p_{i+d}, and thus, every configuration of e is symmetric; *i.e.*, for all $1 \leq i \leq n$ and for all configurations γ_j, $j \geq 0$, of e, all variables of p_i and p_{i+d} have the same value. Eventually, a terminal configuration γ_T is reached. Let p_ℓ be the elected leader in γ_T; thus $\gamma_T(p_\ell).isLeader = \text{TRUE}$. But $\gamma_T(p_{\ell+d}).isLeader$ also, which contradicts the uniqueness of the leader in a solution, since $p_{\ell+d} \neq p_\ell$. ■

Class $\mathcal{K}_k$, $k \geq 2$, contains symmetric rings, *e.g.*, see Figure 3.1. Hence, we have:



(a) Ring in $\mathcal{K}_3$. **(b)** Ring in $\mathcal{K}_4$.

Figure 3.1 – Examples of symmetric ring networks in $\mathcal{K}_k$.



(a) Ring $\mathcal{R}_n$ **(b)** Ring $\mathcal{R}_{n,k}$

Figure 3.2 – Illustration of the proof of Lemma 3.1. In gray, the processes of $\mathcal{R}_{n,k}$ that can have received information from q_{kn} within $t \geq 0$ time units.

Theorem 3.2

For any $k \geq 2$, there is no algorithm that solves message-terminating leader election for $\mathcal{K}_k$.

3.3.2 Lower Bounds on Execution Time for Classes $\mathcal{U}^* \cap \mathcal{K}_k$ and $\mathcal{A} \cap \mathcal{K}_k$, with $k \geq 2$

Lemma 3.1

Let $k \geq 2$ and Alg *be an algorithm that solves the process-terminating leader election for $\mathcal{U}^* \cap \mathcal{K}_k$. $\forall \mathcal{R} \in \mathcal{K}_1$, the synchronous execution of* Alg *in $\mathcal{R}$ lasts at least $1 + (k-2)\,n$ time units, where n is the number of processes.*

Proof: Let $k \geq 2$ and Alg be a process-terminating leader election algorithm for $\mathcal{U}^* \cap \mathcal{K}_k$.

Let $\mathcal{R}_n \in \mathcal{K}_1$ be a ring of n processes, noted $p_0, \ldots, p_{n-1}$ with distinct labels $\ell_0, \ldots, \ell_{n-1}$ respectively, see Figure 3.2a. Since $\mathcal{K}_1 \subseteq \mathcal{U}^* \cap \mathcal{K}_k$, Alg is correct for $\mathcal{R}_n$ and so, the synchronous execution $e = (\gamma_i)_{i \geq 0}$ of Alg on $\mathcal{R}_n$ is finite and a process is elected.

Let T be the execution time of e: within T time units in e, $p_L.isLeader$ becomes TRUE for some $0 \leq L \leq n-1$, *i.e.*, p_L is the leader in the terminal configuration γ_T of e.

We now build the ring $\mathcal{R}_{n,k} \in \mathcal{U}^* \cap \mathcal{K}_k$ of $kn+1$ processes, $q_0, \ldots, q_{kn}$, with labels consisting of the sequence $\ell_0, \ldots, \ell_{n-1}$ repeated k times, followed by a single label $X \notin \{\ell_0, \ldots, \ell_{n-1}\}$, see Figure 3.2b. Let $e' = (\gamma'_i)_{i \geq 0}$ be the synchronous execution of ALG on $\mathcal{R}_{n,k}$. Since $\mathcal{R}_{n,k} \in \mathcal{U}^* \cap \mathcal{K}_k$, ALG is correct on $\mathcal{R}_{n,k}$ so e' is finite and there is no configuration along e' such that two processes declare themselves leader.

By construction, after $t \geq 0$ time units, only the processes q_i, with $i \in \{0, \ldots, t-1\}$, can have received information from process q_{kn} of label X, see the gray zone on Figure 3.2b. Hence, we have the following property on e':

$(*)$ For every $j \in \{0, ..., kn-1\}$, for every $t \geq 0$, if $t \leq j$, then the state of q_j in γ'_t is identical to the state of $p_{j \bmod n}$ in γ_t.

Assume, by contradiction, that $T \leq (k-2)n$. Let $j_1 = (k-2)n+L$ and $j_2 = (k-1)n+L$. Since $L \in \{0, ..., n-1\}$, we have $j_1, j_2 \in \{0, ..., kn-1\}$, hence $T \leq j_1 < j_2$. Moreover, $j_1 \bmod n = j_2 \bmod n = L$. So, by $(*)$ the states of q_{j_1} and q_{j_2} in γ'_T are identical to the state of p_L in γ_T: in particular, $\gamma'_T(q_{j_1}).isLeader = \gamma'_T(q_{j_2}).isLeader =$ TRUE. This contradicts the fact that ALG is a process-terminating leader election algorithm for $R_{n,k}$. (Bullet 5 of the specification is violated in γ'_T, see p. 43.) Hence, the execution time T of the synchronous execution of ALG in R_n is greater than $(k-2)n$. ■

Since $\mathcal{K}_1 \subseteq \mathcal{U}^* \cap \mathcal{K}_k$, follows:

Corollary 3.1

Let $k \geq 2$. The time complexity of any algorithm that solves the process-terminating leader election for $\mathcal{U}^* \cap \mathcal{K}_k$ is $\Omega(k\,n)$ time units, where n is the number of processes.

Furthermore, by definition $\mathcal{U}^* \subseteq \mathcal{A}$, and so:

Corollary 3.2

Let $k \geq 2$. The time complexity of any algorithm that solves the process-terminating leader election for $\mathcal{A} \cap \mathcal{K}_k$ is $\Omega(k\,n)$ time units, where n is the number of processes.

3.3.3 Classes $\mathcal{U}^*$ and $\mathcal{A}$

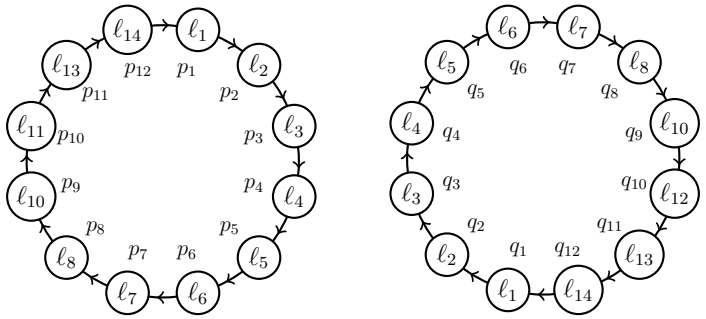
Theorem 3.3

There is no algorithm that solves the process-terminating leader election for $\mathcal{U}^*$.

Proof: Suppose ALG is an algorithm for $\mathcal{U}^*$. Let $\mathcal{R}_n$ be a ring network of $\mathcal{K}_1$ with n processes. Let e be the synchronous execution of ALG on $\mathcal{R}_n$: as $\mathcal{K}_1 \subseteq \mathcal{U}^*$, ALG is correct for $\mathcal{R}_n$ and, consequently, e is finite. Let T be the number of steps of e. We can fix some $k \geq 2$ such that $1 + (k-2)\,n > T$.

Since $(\mathcal{U}^* \cap \mathcal{K}_k) \subseteq \mathcal{U}^*$, ALG is correct for $\mathcal{U}^* \cap \mathcal{K}_k$. By Lemma 3.1, $T \geq 1 + (k-2)n$, a contradiction. ■

Since by definition $\mathcal{U}^* \subseteq \mathcal{A}$, Theorem 3.3 implies the following theorem.



(a) $\mathcal{R}_I$ **(b)** $\mathcal{R}_J$

Figure 3.3 – Ring networks $\mathcal{R}_I$ and $\mathcal{R}_J$ where $m = 7$, $I = \ell_8, \ell_{10}, \ell_{11}, \ell_{13}, \ell_{14}$, and $J = \ell_8, \ell_{10}, \ell_{12}, \ell_{13}, \ell_{14}$ used in the proof of Theorem 3.5.

Theorem 3.4

There is no algorithm that solves the process-terminating leader election for $\mathcal{A}$.

3.3.4 Lower Bounds on the Amount of Exchanged Information

The algorithm proposed by Peterson [Pet82] to solve leader election in identified ring networks has message complexity $O(n \log n)$ and each message contains $\Theta(b)$ bits, *i.e.*, the amount of exchanged information is $O(b\, n \log n)$. As commonly done in the literature, we can assume that $b = \Theta(\log n)$, so $O(b\, n \log n) = O(n \log^2 n)$. Then, as the algorithm of Peterson applies for $\mathcal{U}^* \cap \mathcal{K}_1$, we might expect that there exists a leader election algorithm for class $\mathcal{U}^* \cap \mathcal{K}_k$ whose required amount of exchanged information is $O(k\ n\ \log^2 n)$. Theorem 3.5 shows that, when k is fixed, the minimum amount of exchanged bits needed to solve leader election in the worst case is greater than what we might expect when n is large.

Theorem 3.5

Let $k \geq 2$. For any message-terminating leader election algorithm Alg *for $\mathcal{U}^* \cap \mathcal{K}_k$, there exists executions of* Alg *during which $\Omega(kn + n^2)$ bits are exchanged, where n is the number of processes.*

Proof: Let $k \geq 2$. Let Alg be a message-terminating leader election algorithm for $\mathcal{U}^* \cap \mathcal{K}_k$. Let $m \geq 2$ and let $n = 2m$. Let $\ell_1, \ldots, \ell_n$ be distinct labels.

Let $\mathcal{L}$ be the set of non-empty proper subsequences of $(\ell_{m+1}, \ldots, \ell_n)$, *i.e.*, $() \notin \mathcal{L}$ and $(\ell_{m+1}, \ldots, \ell_n) \notin \mathcal{L}$. For any $I \in \mathcal{L}$, $\mathcal{R}_I$ is the ring network containing $m + |I|$ processes, denoted p_1, p_2, $\ldots$, p_m, p_{m+1}, $\ldots$, $p_{m+|I|}$, whose label sequence is $\Lambda_I = \ell_1, \ell_2, \ldots, \ell_m, I$. More precisely, in $\mathcal{R}_I$, for every $i \in \{1, \ldots, m\}$, $p_i.id = \ell_i$ and for every $j \in \{1, \ldots, |I|\}$, $p_{m+j}.id = I[j]$ (the j^{th} element of I). The ring network $\mathcal{R}_I$ for $m = 7$ and $I = \ell_8, \ell_{10}, \ell_{11}, \ell_{13}, \ell_{14}$ is illustrated on Figure 3.3a.

Let $\mathfrak{R} = \{\mathcal{R}_I : I \in \mathcal{L}\}$. Notice that $|\mathfrak{R}| = 2^m - 2$ (since the empty sequence and $\{\ell_{m+1}, \ldots, \ell_n\}$ are not in $\mathcal{L}$). Furthermore, every label in $\mathcal{R}_I$ is unique so $\mathcal{R}_I \in \mathcal{U}^* \cap \mathcal{K}_k$. Hence, Alg is correct for every $\mathcal{R}_I \in \mathfrak{R}$.

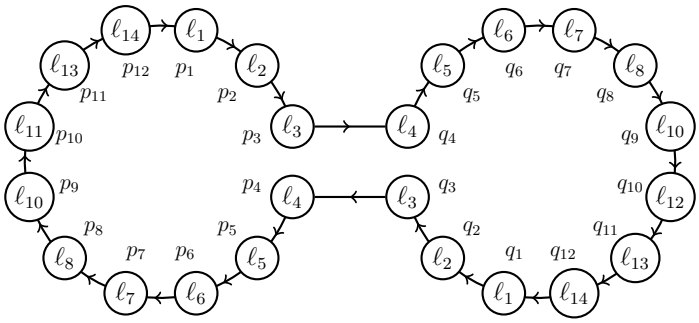


Figure 3.4 – Ring network $\mathcal{R}^p_{I\#J}$ where $m = 7$, $p = 3$, $I = \ell_8, \ell_{10}, \ell_{11}, \ell_{13}, \ell_{14}$, and $J = \ell_8, \ell_{10}, \ell_{12}, \ell_{13}, \ell_{14}$ used in the proof of Theorem 3.5.

For every $I, J \in \mathcal{L}$, let $\mathcal{R}_{I\#J}$ be the ring network containing $2m + |I| + |J|$ processes whose label sequence is $\Lambda_I \Lambda_J$. $\mathcal{R}_{I\#J}$ can be obtained from $\mathcal{R}_I$ and $\mathcal{R}_J$ as follows (we denote by $q_1, \ldots, q_{m+|J|}$ the processes of $\mathcal{R}_J$ to avoid confusion). For some $p \in \{1, \ldots, m-1\}$, we obtain $\mathcal{R}^p_{I\#J}$ when we join $\mathcal{R}_I$ and $\mathcal{R}_J$ by removing edges (p_p, p_{p+1}) and (q_p, q_{p+1}) and replacing them by edges (p_p, q_{p+1}) and (q_p, p_{p+1}). Figure 3.4 shows the ring network $\mathcal{R}^p_{I\#J}$ for $p = 3$, $I = \ell_8, \ell_{10}, \ell_{11}, \ell_{13}, \ell_{14}$, and $J = \ell_8, \ell_{10}, \ell_{12}, \ell_{13}, \ell_{14}$ obtained by joining $\mathcal{R}_I$ (see Figure 3.3a) and $\mathcal{R}_J$ (see Figure 3.3b).

Claim 1: For every $p \in \{1, \ldots, m-1\}$, if $I \neq J$, then $\mathcal{R}^p_{I\#J} \in \mathcal{U}^* \cap \mathcal{K}_k$.

Proof of the claim: No label appears more than twice in $\mathcal{R}^p_{I\#J}$ so $\mathcal{R}^p_{I\#J} \in \mathcal{K}_k$. Then, without loss of generality, assume $|I| \leq |J|$. So, there is some $\ell_j \in J \backslash I$ and ℓ_j is a unique label in $\mathcal{R}^p_{I\#J}$. Hence, $\mathcal{R}^p_{I\#J} \in \mathcal{U}^*$. □

For every $I \in \mathcal{L}$, let e_I be the synchronous execution of Alg on $\mathcal{R}_I$. For each $p \in \{1, \ldots, m\}$, let $\sigma_{I,p}$ be the stream (sequence) of bits sent by p_p to p_{p+1} during e_I.

Claim 2: For any $I, J \in \mathcal{I}$ and any $p \in \{1, \ldots, m-1\}$, if $\sigma_{I,p} = \sigma_{J,p}$, then $I = J$.

Proof of the claim: Let $I, J \in \mathcal{I}$ and $p \in \{1, \ldots, m-1\}$. Assume that $\sigma_{I,p} = \sigma_{J,p}$. Let $\mathcal{R}^p_{I\#J}$ the ring network obtained by joining $\mathcal{R}_I$ and $\mathcal{R}_J$ at the edges (p_p, p_{p+1}) and (q_p, q_{p+1}). On Figure 3.4, $p = 3$. Let $e^p_{I\#J}$ be the synchronous execution of Alg on $\mathcal{R}^p_{I\#J}$.

First, we show by induction on the steps of $e^p_{I\#J}$ that $\forall x \geq 1$, every process p_i, $i \in \{1, \ldots, m + |I|\}$ (respectively, q_j, $j \in \{1, \ldots, m + |J|\}$) sends the same bits during the x^{th} step of $e^p_{I\#J}$ than in the x^{th} step of e_I (respectively, e_J).

Base Case: Alg is a deterministic algorithm and every process p_i (respectively, q_j) has the same initial state (in particular, the same ID) in $e^p_{I\#J}$ than in e_I (respectively, e_J). Hence every process p_i (respectively, q_j) sends the same bits during the first step of $e^p_{I\#J}$ than during the first step of e_I (respectively, e_J).

Induction Step: Assume that every process p_i, $i \in \{1, \ldots, m + |I|\}$ (respectively, q_j, $j \in \{1, \ldots, m + |J|\}$) sends the same bits during the x^{th} step of $e^p_{I\#J}$ than in the x^{th} step of e_I (respectively, e_J), $x \geq 1$. Consider the $(x+1)^{\text{th}}$ step of $e^p_{I\#J}$.
Every process p_i, $i \in \{1, \ldots, p\} \cup \{p+2, m + |I|\}$, (respectively, q_j, $j \in \{1, \ldots, p\} \cup \{p+2, \ldots, m + |J|\}$) has the same predecessor in $\mathcal{R}^p_{I\#J}$ than in $\mathcal{R}_I$ (respectively, $\mathcal{R}_J$). By induction hypothesis, this predecessor sends

the same bits during the x^{th} step of $e^p_{I\#J}$ than during the x^{th} step of e_I (respectively, e_J).
Now, the only processes that do not have the same predecessor in $\mathcal{R}^p_{I\#J}$ than in $\mathcal{R}_I$ or $\mathcal{R}_J$ are p_{p+1} and q_{p+1}. By induction hypothesis, their predecessor, respectively q_p and p_p, send the same bits during the x^{th} step of respectively $e^p_{I\#J}$ than during the x^{th} step of respectively e_J and e_I. Furthermore, $\sigma_{I,p} = \sigma_{J,p}$ so they send the exact same bits.
Hence, every process receives the same bits and so send the same bits (since the algorithm is deterministic) during the $(x+1)^{\text{th}}$ step of $e^p_{I\#J}$ than during the $(x+1)^{\text{th}}$ step of e_I or e_J.

Hence, the processes cannot distinguish $e^p_{I\#J}$ and e_I or e_J. So both the process that declares itself leader in e_I and the one that declares itself leader in e_J also declare themselves leader in $e^p_{I\#J}$. Now, assume by contradiction that $I \neq J$. By Claim 1, $\mathcal{R}^p_{I\#J} \in \mathcal{U}^* \cap \mathcal{K}_k$. Since two processes declare themselves leader in $e^p_{I\#J}$, we have a contradiction with the correctness of ALG for class $\mathcal{U}^* \cap \mathcal{K}_k$. □

The rest of the proof is based on a counting argument, *i.e.*, there are not enough bit streams to distinguish the rings of $\mathfrak{R}$, unless those streams have length $\Omega(n^2)$.

Let $a = m - 2$. Recall that a set of cardinality m has 2^m subsets including $2^m - 2$ non-empty proper subsets. The number of bit streams of length at most a is:

$$\sum_{l=1}^{a} 2^l = 2^{a+1} - 1 = 2^{m-1} - 1 < 2^m - 2$$

Let $p \in \{1, \ldots, m-1\}$. Let $\mathcal{L}_p = \{I \in \mathcal{L} : |\sigma_{I,p}| \leq a\}$. By Claim 2, $\mathcal{L}_p$ has cardinality less than $2^m - 2$, so there exists some $I \in \mathcal{L}$ that is not a member of $\mathcal{L}_p$, for any $p \in \{1, \ldots, m-1\}$. Hence, $\Omega(m\ a) = \Omega(n^2)$ bits are exchanged during the synchronous execution of ALG on $\mathcal{R}_I$.

Now, at least one message must be exchanged at each step, so, by Corollary 3.1, there exists executions of ALG where $\Omega(kn + n^2)$ bits are exchanged. ■

Since $\mathcal{U}^* \cap \mathcal{K}_k \subseteq \mathcal{A} \cap \mathcal{K}_k$, follows:

Theorem 3.6

Let $k \geq 2$. For any message-terminating leader election algorithm ALG *for $\mathcal{A} \cap \mathcal{K}_k$, there exists executions of* ALG *during which $\Omega(kn + n^2)$ bits are exchanged, where n is the number of processes.*

3.4 Algorithm U_k of Leader Election in $\mathcal{U}^* \cap \mathcal{K}_k$

In this section, we present a process-terminating leader election algorithm U_k for class $\mathcal{U}^* \cap \mathcal{K}_k$, for any $k \geq 1$, see Algorithm 1.

Algorithm 1 – Actions of Process p in Algorithm U_k.

Inputs.

- $p.id \in \mathfrak{id}$

Variables.

- $p.init \in \mathbb{B} = \{\text{TRUE}, \text{FALSE}\}$, initially TRUE
- $p.active \in \mathbb{B}$, initially TRUE
- $p.count \in \{0, \ldots, k+1\}$, initially 0
- $p.leader \in \mathfrak{id}$
- $p.isLeader \in \mathbb{B}$, initially FALSE
- $p.done \in \mathbb{B}$, initially FALSE

Actions.

A1	::	$p.init$	$\rightarrow$	$p.init := \text{FALSE}$ **send** $\langle p.id, 0\rangle$
A2	::	$\neg p.init \wedge p.active \wedge$ **rcv** $\langle x, c\rangle \wedge x \neq p.id$ $\wedge\ (p.count = 0 \vee c > p.count)$	$\rightarrow$	**send** $\langle x, c\rangle$
A3	::	$\neg p.init \wedge p.active \wedge$ **rcv** $\langle x, c\rangle \wedge x > p.id$ $\wedge\ c = p.count \wedge c \geq 1$	$\rightarrow$	**send** $\langle x, c\rangle$
A4	::	$\neg p.init \wedge p.active \wedge$ **rcv** $\langle x, c\rangle \wedge x = p.id$ $\wedge\ c = p.count \wedge c \leq k-1$	$\rightarrow$	$p.count := c+1$ **send** $\langle x, c+1\rangle$
(Deactivation)				
A5	::	$\neg p.init \wedge p.active \wedge$ **rcv** $\langle x, c\rangle \wedge x \neq p.id$ $\wedge\ c < p.count$	$\rightarrow$	$p.active := \text{FALSE}$ **send** $\langle x, c\rangle$
A6	::	$\neg p.init \wedge p.active \wedge$ **rcv** $\langle x, c\rangle \wedge x < p.id$ $\wedge\ c = p.count \wedge c \geq 1$	$\rightarrow$	$p.active := \text{FALSE}$ **send** $\langle x, c\rangle$
(Passive Processes)				
A7	::	$\neg p.init \wedge \neg p.active \wedge$ **rcv** $\langle x, c\rangle \wedge x \neq p.id \wedge c \leq k$	$\rightarrow$	**send** $\langle x, c\rangle$
A8	::	$\neg p.init \wedge \neg p.active \wedge$ **rcv** $\langle x, c\rangle \wedge x = p.id$	$\rightarrow$	(nothing)
(Ending Phase)				
A9	::	$\neg p.init \wedge p.active \wedge$ **rcv** $\langle x, k\rangle \wedge x = p.id$ $\wedge\ p.count = k$	$\rightarrow$	$p.isLeader := \text{TRUE}$ $p.leader := p.id$ $p.done := \text{TRUE}$ $p.count := k+1$ **send** $\langle x, k+1\rangle$
A10	::	$\neg p.init \wedge \neg p.active \wedge$ **rcv** $\langle x, k+1\rangle$	$\rightarrow$	$p.leader := x$ $p.done := \text{TRUE}$ **send** $\langle x, k+1\rangle$ (halt)
A11	::	$\neg p.init \wedge p.active \wedge$ **rcv** $\langle x, k+1\rangle \wedge x = p.id$ $\wedge\ p.count = k+1$	$\rightarrow$	(halt)

3.4.1 Overview of U_k

U_k elects the process of minimum unique label to be the leader, namely the process L such that $L.id = \min\{p.id : p \in V \wedge \text{mlty}(p.id) = 1\}$. In U_k, each process p has the following variables.

1. $p.id \in \mathfrak{id}$, (constant) input of unspecified *label type*, the label of p.

2. $p.init$, Boolean, initially TRUE.

3. $p.active$, Boolean, which indicates that p is *active*. If $\neg p.active$, we say p is *passive*. Initially, all processes are active, and when U_k is done, the leader is the only active process. A passive process never becomes active.

4. $p.count$, an integer in the range $0 \ldots k+1$. Initially, $p.count = 0$. $p.count$ will give to p a rough estimate of the predominance of its label in the ring.

5. $p.leader$, of label type. When U_k is done, $p.leader = L.id$.

6. $p.isLeader$, Boolean, initially FALSE, follows the problem specification: eventually, $L.isLeader$ becomes TRUE and remains TRUE, while, for all $p \neq L$, $p.isLeader$ remains FALSE for the entire execution.

7. $p.done$, Boolean, initially FALSE, follows the problem specification: eventually, $p.done$ = TRUE for all p. $p.done$ means that p knows a leader has been elected; once true, it will never become false.

U_k uses only one kind of message. Each message is the forwarding of a *token* which is generated at the initialization of the algorithm, and is of the form $\langle x, c\rangle$, where x is the label of the originating process, and c is a *counter*, an integer in the range $0 \ldots k+1$, initially zero.

The explanation below is illustrated by the example in Figure 3.5. The fundamental idea of U_k is that a process becomes passive, *i.e.*, is no more candidate for the election, if it receives a message that proves its label is not unique or is not the smallest unique label.

Counter Increments. Initially, every process initiates a token with its own label and counter zero (see (a)). No tokens are initiated afterwards. The token continuously moves around the ring – every time it is forwarded, its counter and the local counter of the process are incremented if the forwarding process has the same label as the token (*e.g.*, Step (a)↦(b)). Thus, if the message $\langle x, c\rangle$ is in a channel, that token was initiated by a process whose label is x, and has been forwarded c times by processes whose labels are also x. The token could also have been forwarded any number of times by processes with labels which are not x. Thus, the counter in a message is a rough estimate of the predominance of its label in the ring.

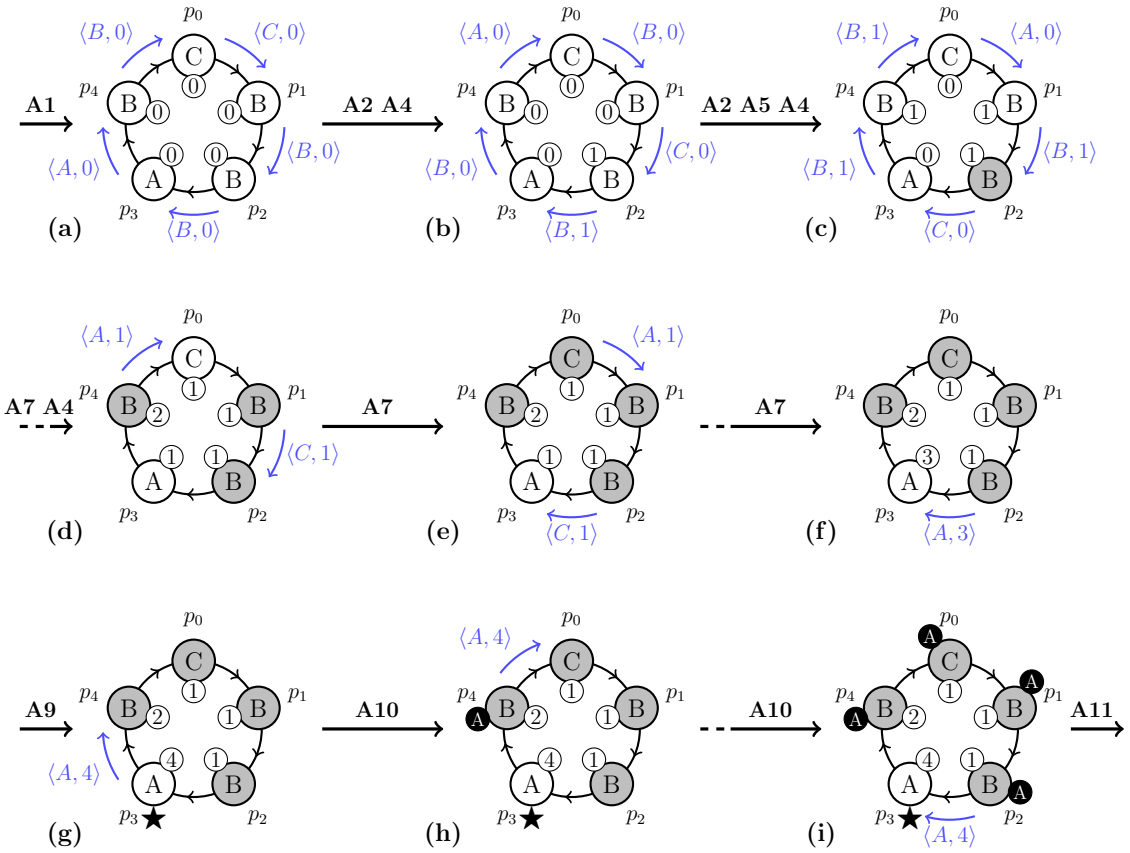


Figure 3.5 – Example of execution of U_k where $k = 3$. The counter of a process is in the white bubble next to the corresponding node. Gray nodes are passive. $p.isLeader = \text{TRUE}$ if there is a star next to the node. The black bubble contains the elected label, $p.leader$.

Non-unique Label Elimination. If a process p receives a message whose counter is less than $p.count$, and $p.count \geq 1$, this proves its label is not unique since its counter grows faster than the one of another label. In this case, p executes **A5**-action and becomes passive (*e.g.*, Step (b)↦(c)).

Since the counter in the message initiated by L is never incremented, except by L itself, every process whose label is not unique becomes passive during the first traversal of $\langle L.id, 0\rangle$.

Non-lowest Unique Label Elimination. Similarly, if a process p has a unique label but not the smallest one, it will become passive executing **A6**-action when p receives a message with the same non-zero counter but a label lower than $p.id$ (*e.g.*, Step (d)↦(e)).

This happens at the latest when the process receives the message $\langle L.id, 1\rangle$, *i.e.*, before the second time L receives its own token. So, after the token of L has made two traversals of the ring, it is the only surviving token (the others are consumed by **A8**-action) and every process but L is passive.

Termination Detection. The execution continues until the leader L has seen its own label return to it k times, otherwise L cannot be sure that what it has seen is not part

of a larger ring instead of several rounds of a small ring. Then, L designates itself as leader by **A9**-action (see Step (f)↦(g)) and its token does a last traversal of the ring to inform the other processes of its election (*e.g.*, Step (g)↦(h)). The execution ends when L receives its token after $k+2$ traversals (see (i)).

3.4.2 Correctness and Complexity Study

To prove the correctness of U_k (Theorem 3.7), we first prove some results on the counters of the messages (Lemma 3.2). Then, Lemmas 3.3-3.5 prove the properties of the different phases of U_k (Lemma 3.7). Finally, Theorem 3.8 proves its complexity.

In the following proofs, we write #hop(m) for the number of hops, *i.e.*, how many times the token has been received, made so far by the token associated to the message m. Notice that #hop(m) is always of the form $an+b$ where $a \geq 0$ is the number of complete traversal realized by m and $0 \leq b < n$ is the shift of the position of m on the ring compared to the position of the initiator of m.

Lemma 3.2

Let $\gamma \mapsto \gamma'$ be a step. Suppose a message $\langle x, c\rangle$ such that #hop$(\langle x, c\rangle) = an+b$ in γ with $a \geq 0$ and $0 \leq b < n$ is sent in $\gamma \mapsto \gamma'$. Then:

a). $c \geq a$,

b). *if x is a unique label, then $c = a$, and*

c). *if x is a not a unique label and $a \geq 1$, then $c > a$.*

Proof: Let p be the process which originated the token currently carried by the message m.

The token has made a complete traversals of the ring, and has visited p a times, hence its counter has been incremented at least a times. This proves (a).

If p is the only process with label x, then the counter has not otherwise been incremented, and we have (b).

Suppose x is not a unique label, and $a \geq 1$. There are at least two processes with ID x. The token has made at least a full traversals, and thus has been sent by processes of ID x at least $2a$ times. Starting at zero, c has been incremented at least $2a$ times, hence $c \geq 2a > a$. We have (c). ■

For the next lemma, we recall that a process can become passive only by executing **A5** or **A6**-action.

Lemma 3.3

L never becomes passive.

Proof: By contradiction, assume L becomes passive during some step $\gamma \mapsto \gamma'$. Then L executes **A5** or **A6**-action, receiving the message $\langle x, c\rangle$ for some $x \neq L.id$. Since the

label of L is unique, the token it initiated is still circulating in the ring in γ (it cannot be discarded except by L if it becomes passive). Moreover, since $x \neq L.id$, $\#\text{hop}(\langle x, c\rangle)$ is not a multiple of n in γ. Let $\#\text{hop}(\langle x, c\rangle) = an + b$ in γ, where $a \geq 0$ and $1 \leq b < n$. Since the links are FIFO, the token initiated by L has made a full circuits during the prefix of execution leading to γ, and $\gamma(L).count = a$. We now consider two cases.

- **Case 1:** x is a unique label. By Lemma 3.2(b), $c = a = L.count$. Thus, L cannot execute **A5**-action; since $L.id < x$, L cannot execute **A6**-action. Contradiction.
- **Case 2:** x is not unique. *N.b.* $L.count = a$ in γ. If $a = 0$, then L is not enabled to execute either action. If $a \geq 1$, then $c > a$ by Lemma 3.2(c), contradiction. ■

We define an L-tour as follows. The first L-tour of an execution $e = (\gamma_i)_{i\geq 0}$ is the minimum prefix of execution that terminates by a step $\gamma_j \mapsto \gamma_{j+1}$ where L receives (and treats) a message tagged with its own label for the first time. The second L-tour is the first L-tour of the execution suffix $e' = (\gamma_i)_{i\geq j}$ starting in γ_j, and so forth. From Lemma 3.3, the code of the algorithm, and the fact that the label of L is unique, we have:

Corollary 3.3

Any execution contains exactly $k + 2$ complete L-tours.

Lemma 3.4

For any process p, if $p \neq L$ and $p.id$ is a unique label, then p becomes passive within the first two L-tours.

Proof: Let $x = p.id$. By definition of L, $x > L.id$. Let $d = \|L, p\|$. Suppose by contradiction that x does not become passive during the first two L-tours (which are defined, Corollary 3.3). The token t initiated by L is received by p during the first (resp. second) L-tour while $\#\text{hop}(t) = d$ (resp. $\#\text{hop}(t) = n + d$). p receives the token it initiates exactly once before receiving $t = \langle x, c\rangle$ in $\gamma \mapsto \gamma'$ during the second L-tour. So, as x is unique, we have $p.count = 1$ in γ. Now, $c = 1$ in γ (Lemma 3.2(a)). Thus, p becomes passive by executing **A6**-action in $\gamma \mapsto \gamma'$, contradiction. ■

Lemma 3.5

If z is a non-unique label, then all processes of label z become passive within the first two L-tours.

Proof: Let $m \geq 2$ be the multiplicity of z, and let $\mathcal{P}[z] = \{x_1, x_2, \ldots x_m\}$ be the sequence of processes of label z in clockwise order from L.

Claim 1: Any process x_i with $i \neq 1$ receives the token initiated by x_{i-1} of the form $\langle z, 0\rangle$ during the first L-tour before receiving $\langle L.id, 0\rangle$.

Proof of the claim: L is not between x_{i-1} and x_i, and no process between x_{i-1} and x_i can stop the message $\langle z, 0\rangle$ initiated by x_{i-1}. So, x_i will receive $\langle z, 0\rangle$ before receiving $\langle L.id, 0\rangle$ during the first L-tour. □

Claim 2: x_1 receives $\langle z, 0\rangle$ and then $\langle z, 1\rangle$ during the first two L-tours, both of them before receiving $\langle L.id, 1\rangle$.

Proof of the claim: No process between x_m and x_1 can stop the message $\langle z, 0\rangle$ initiated by x_m. Then, by Claim 1, x_m receives a message $\langle z, 0\rangle$, while satisfying $x_m.count = 0$. So, x_m sends $\langle z, 1\rangle$ after $\langle z, 0\rangle$, but before receiving $\langle L.id, 0\rangle$. Again, no process between x_m and x_1 can stop that message. So, x_1 receives $\langle z, 0\rangle$ and $\langle z, 1\rangle$ before receiving $\langle L.id, 1\rangle$, *i.e.*, during the first two L-tours. □

Claim 3: Every process x_i with $i \neq 1$ receives $\langle z, 1\rangle$ during the first two L-tours before receiving $\langle L.id, 1\rangle$.

Proof of the claim: The first time x_{i-1} receives $\langle z, 0\rangle$ is before x_{i-1} receives $\langle L.id, 1\rangle$ in the first two L-tours, by Claims 1 and 2. In that step, x_{i-1} sends $\langle z, 1\rangle$. No process between x_{i-1} and x_i can stop that message. So, x_i receives $\langle z, 1\rangle$ during the first two L-tours before receiving $\langle L.id, 1\rangle$. □

By Claims 2 and 3, each x_i receives the message $\langle z, 1\rangle$ during the first two L-tours before receiving $\langle L.id, 1\rangle$. Consider the first time x_i receives such a message. Then, $x_i.count = 1$. Either x_i is already passive and we are done, or $x_i.count$ is set to 2. Hence, when receiving $\langle L.id, 1\rangle$ during the first two L-tours, x_i executes **A5**-action and we are done. ■

Lemma 3.6

For any process p, if $p \neq L$, then p never executes **A9***-action.*

Proof: Assume, by the contradiction, that some process $p \neq L$ eventually executes **A9**-action. Let $x = p.id$. Then, p successively receives $\langle x, 0\rangle, \ldots, \langle x, k\rangle$ so that $p.active \wedge p.count = k$ holds when p receives $\langle x, k\rangle$. Notice also that p also receives $\langle L.id, 0\rangle$ and $\langle L.id, 1\rangle$, by Corollary 3.3.

First, p does not receive $\langle L.id, 0\rangle$ after $\langle x, k\rangle$, because otherwise p received at least $k+1$ messages tagged with label x during the first L-tour, which is impossible since the multiplicity of x is at most k and the links are FIFO.

Assume now that p receives $\langle L.id, 0\rangle$ before $\langle x, k\rangle$ but after $\langle x, 0\rangle$. Then, p is deactivated by **A5**-action when it receives $\langle L.id, 0\rangle$ because $p.count > 0$ and so before receiving $\langle x, k\rangle$, a contradiction.

So, p receives $\langle L.id, 0\rangle$ before $\langle x, 0\rangle$. Similarly, p does not receive $\langle L.id, 1\rangle$ after $\langle x, k\rangle$, because otherwise p received at least $k+1$ messages tagged with label x during the first L-tour. Then, p does not receive $\langle L.id, 1\rangle$ before $\langle x, 0\rangle$ because otherwise p does not receive any message tagged with x during the first L-tour, now it receives at least $\langle x, 0\rangle$ during the first L-tour from either its first predecessor with same label, or itself (if x is unique in the ring).

If p receives $\langle L.id, 1\rangle$ before $\langle x, 1\rangle$, then x is unique in the ring and when p receives $\langle L.id, 1\rangle$, p is deactivated by **A6**-action, and so before receiving $\langle x, k\rangle$, a contradiction.

Finally, if $k > 1$ and if p receives $\langle L.id, 1\rangle$ after $\langle x, 1\rangle$ but before $\langle x, k\rangle$, then p is deactivated by **A5**-action when it receives $\langle L.id, 1\rangle$, because $1 < p.count \leq k$. Hence, again, p is deactivated before receiving $\langle x, k\rangle$, a contradiction. ■

Lemma 3.7

In any execution of U_k:

a). *For every process $p \neq L$, $p.active$ becomes* FALSE *within the first two L-tours.*

b). *For every process $p \neq L$, p never executes* **A9***-action.*

c). *L executes* **A9***-action after exactly $k+1$ L-tours. In this action $L.leader := L$, $L.isLeader :=$* TRUE*, and $L.done :=$* TRUE*.*

d). *For every process $p \neq L$ is a process, p executes* **A10***-action during the $(k+2)^{nd}$ L-tour. In this action $p.leader := L$ and $p.done :=$* TRUE*.*

e). *L executes* **A11***-action after exactly $k+2$ L-tours, and that is the last action of the execution.*

Proof: Part (a) follows from Lemmas 3.4 and 3.5.

Part (b) is Lemma 3.6.

Parts (c)–(e) follow from Corollary 3.3. The token initialized by L circles the ring $k+2$ times, each time incrementing $L.count$ once. At the end of the $(k+1)^{\text{st}}$ traversal, L executes **A9**-action, electing itself to be the leader. The message $\langle L.id, k+1\rangle$ then circles the ring, informing all other processes that L has been elected. Those latter processes halt after forwarding this message. When that final message reaches L, the execution is over. ■

The main theorem of this subsection, Theorem 3.7 below, follows immediately from Lemma 3.7.

Theorem 3.7

U_k solves the process-terminating leader election for $\mathcal{U}^* \cap \mathcal{K}_k$, for every given $k \geq 1$.

Theorem 3.8

U_k has time complexity at most $n(k+2)$, has message complexity $O(n^2 + kn)$, and requires $\lceil \log(k+1) \rceil + 2b + 4$ bits in each process.

Proof: Time complexity follows from Lemma 3.7. Space complexity follows from the definition of U_k.

Consider now the message complexity of U_k. All tokens, except the one initiated by L, vanish during the three first L-tours, by Lemma 3.7(a). Consequently, only the token initiated by L circulates during the $k-1$ last L-tours. Hence, we obtain a message complexity of $O(n^2 + kn)$ ($O(n^2)$ for messages transmitted during the 3 first L-tours, and kn, with $k \leq n$ for the unique token circulating during the $k-1$ last L-tours). ■

3.5 Algorithm A_k of Leader Election in $\mathcal{A} \cap \mathcal{K}_k$

We now give a solution, Algorithm A_k, to the process-terminating leader election for the class $\mathcal{A} \cap \mathcal{K}_k$, for fixed $k \geq 1$. A_k is based on the following observation. Consider a ring $\mathcal{R}$ of $\mathcal{A} \cap \mathcal{K}_k$ with n processes. As $\mathcal{R}$ is asymmetric, any two processes in $\mathcal{R}$ can be distinguished by examining all labels. So, using the lexicographical order, a process can be elected as the leader by examining all labels. Initially, any process p of $\mathcal{R}$ does not know the labels of $\mathcal{R}$, except its own. But, if each process broadcasts its own label clockwise, then any process can learn the labels of all other processes from messages it receives from its left neighbor. In the following, we show that, after examining finitely many labels, a process can decide that it learned (at least) all labels of $\mathcal{R}$ and so can determine whether it is the leader.

3.5.1 Overview of A_k

Sequences of Labels. Given any process p of $\mathcal{R}$, we define $LSeq(p)$, to be the infinite sequence of labels of processes, starting at p and continuing counter-clockwise forever:

$$LSeq(p_i) = p_i.id, p_{i-1}.id, p_{i-2}.id \ldots, \text{ where subscripts are modulo } n.$$

For example, if the ring has three processes where $p_0.id = p_1.id = A$ and $p_2.id = B$, then $LSeq(p_0) = ABAABA\ldots$

For any sequence of labels σ, we define σ^t as the prefix of σ of length t, and $\sigma[i]$, for all $i \geq 1$, as the i^{th} element (starting from the left) of σ.

If σ is an infinite sequence (resp. a finite sequence of length λ), we say that $\pi = \sigma^m$ is a *repeating prefix* of σ if $\sigma[i] = \pi[1 + (i-1) \bmod m]$ for all $i \geq 1$ (resp. for all $1 \leq i \leq \lambda$). Informally, if σ is infinite, then σ is the concatenation $\pi\pi\pi\ldots$ of infinitely many copies of π, otherwise σ is the truncation at length λ of the infinite sequence $\pi\pi\pi\ldots$

Let $srp(\sigma)$ be the *repeating prefix of σ of minimum length.*

As $\mathcal{R}$ is asymmetric, we have:

Lemma 3.8

Let p be a process and $m \in \{2n, \ldots, \infty\}$. The length of $srp(LSeq(p)^m)$ is n.

Proof: Let s be the smallest length of any repeating prefix of σ. $LSeq^n(p)$ is a repeating prefix of σ and thus s is defined, and $s \leq n$.

If $s < n$, then rotation by s is a non-trivial rotational symmetry of $\mathcal{R}$, contradicting the hypothesis that $\mathcal{R}$ is asymmetric. ■

The next lemma shows that any process p can *fully determine* $\mathcal{R}$, *i.e.*, p can determine n, as well as the labeling of $\mathcal{R}$, from any prefix of $LSeq(p)$, provided that prefix contains at least $2k+1$ copies of any label.

Lemma 3.9

Let p be a process, $m > 0$ and ℓ be a label. If $LSeq(p)^m$ contains at least $2k+1$ copies of ℓ, then $\mathcal{R}$ is fully determined by $LSeq(p)^m$.

Proof: We note $\pi = LSeq(p)^m$ and assume that it contains at least $2k+1$ copies of ℓ. First, $m > 2n$. Indeed, there are at most k copies of ℓ in any subsequence of $LSeq(p)$ of length no more than n, by definition of $\mathcal{K}_k$. So, at most $2k$ copies of ℓ in any subsequence of length no more than $2n$. Then, by Lemma 3.8, $srp(\pi) = LSeq(p)^n$. Hence, one can compute $srp(\pi)$: its length provides n and its contents is exactly the counter-clockwise sequence of labels in R, starting from p. ■

True Leader. We define the *true leader* of $\mathcal{R}$ as the process L such that $LSeq(L)^n$ is a *Lyndon word* [Lyn54], *i.e.*, a non-empty string that is strictly smaller in lexicographic order than all of its rotations. In the following, we note $LW(\sigma)$ the rotation of the sequence σ which is a Lyndon word.

In Algorithm A_k (see Algorithm 2), the true leader will be elected. Precisely, in A_k, a process p uses a variable $p.string$ to save a prefix of $LSeq(p)$ at any step: $p.string$ is initially empty and consists of all the labels that p has received during the execution of A_k so far. Lemma 3.9 shows how p can determine the label of the true leader. Indeed, if $p.string$ contains at least $2k+1$ copies of some label, $srp(p.string) = LSeq(p)^n$. If $srp(p.string) = LW(srp(p.string))$, then p is the true leader. Otherwise, the label of the true leader is the first label of $LW(srp(p.string))$, *i.e.*, $LW(srp(p.string))[1]$.

In A_k, we use the function $Leader(\sigma)$ which returns TRUE if the sequence σ contains at least $2k+1$ copies of some label and $srp(\sigma) = LW(srp(\sigma))$, FALSE otherwise.

Overview of A_k. Each process p has six variables. As defined in the specification, p has the variables $p.id$ and $p.leader$ (of label type), and $p.done$ and $p.isLeader$ (Booleans, initially FALSE). p also has a Boolean variable $p.init$, initially TRUE, and the variable $p.string$, as defined above. There are two kinds of messages: $\langle x \rangle$ where x is of label type and $\langle$FINISH$\rangle$.

A_k consists of two phases, which we call the *string growth phase* and the *finishing phase*.

During the string growth phase, each process p builds a prefix of $LSeq(p)$ in $p.string$. First, p initiates a token containing its label, and also initializes $p.string$ to $p.id$ (**A1**-action). The token moves around the ring repeatedly until the end of the string growth phase. When p receives a label, p executes **A2**-action to append it to its string, and sends it to its right neighbor. Thus, each process keeps growing $p.string$.

Eventually, L receives a label x such that $L.string \cdot x$ is long enough for L to determine that it is the leader, see Lemma 3.9 and the definition of function $Leader$. In this case, L executes **A3**-action: L appends $L.string$ with x, ends the string growth phase, initiates the finishing phase by electing itself as leader, and sends the message $\langle$FINISH$\rangle$ to its right neighbor. The message $\langle$FINISH$\rangle$ traverses the ring, informing all

Algorithm 2 – Actions of Process p in Algorithm A_k.

Inputs.

- $p.id \in \mathfrak{id}$

Variables.

- $p.init \in \mathbb{B} = \{\text{TRUE}, \text{FALSE}\}$, initially TRUE
- $p.string$, a sequence of labels, initially empty
- $p.leader \in \mathfrak{id}$
- $p.isLeader \in \mathbb{B}$, initially FALSE
- $p.done \in \mathbb{B}$, initially FALSE

Actions.

A1	::	$p.init$	$\rightarrow$	$p.string := p.id$ $p.init := \text{FALSE}$ **send** $\langle p.id \rangle$
A2	::	$\neg p.init \wedge$ **rcv** $\langle x \rangle \wedge \neg Leader(p.string \cdot x)$	$\rightarrow$	$p.string := p.string \cdot x$ **send** $\langle x \rangle$
A3	::	$\neg p.init \wedge$ **rcv** $\langle x \rangle \wedge Leader(p.string \cdot x)$ $\wedge \neg p.isLeader$	$\rightarrow$	$p.string := p.string \cdot x$ $p.isLeader := \text{TRUE}$ $p.leader := p.id$ $p.done := \text{TRUE}$ **send** $\langle \text{FINISH} \rangle$
A4	::	$\neg p.init \wedge$ **rcv** $\langle \text{FINISH} \rangle \wedge \neg p.isLeader$	$\rightarrow$	$p.leader := LW(srp(p.string))[1]$ $p.done := \text{TRUE}$ **send** $\langle \text{FINISH} \rangle$ (halt)
A5	::	$\neg p.init \wedge$ **rcv** $\langle x \rangle \wedge p.isLeader$	$\rightarrow$	(nothing)
A6	::	$\neg p.init \wedge$ **rcv** $\langle \text{FINISH} \rangle \wedge p.isLeader$	$\rightarrow$	(halt)

processes that the election is over. As each process p receives the message (**A4**-action), it knows that a leader has been elected, can determine its label, $LW(srp(p.string))[1]$, and then halts. Meanwhile, L consumes every token (**A5**-action). When $\langle \text{FINISH} \rangle$ returns to L, it executes **A6**-action and halts, concluding the execution of A_k.

3.5.2 Correctness and Complexity Study

Theorem 3.9

A_k solves the process-terminating leader election for $\mathcal{A} \cap \mathcal{K}_k$, for every given $k \geq 1$.

Proof: Let $M = \max\{\text{mlty}(\ell) : \ell \text{ is a label in } \mathcal{R}\}$ and $m = \lceil (2k+1)/M \rceil\, n$. After receiving at most m messages containing labels (the messages cannot be discarded before the election of a leader, **A5**-action), by Lemma 3.9, every process will know $\mathcal{R}$ completely. Hence, by definition, L can determine that it is the true leader. As soon as L realizes that it is the leader, it will execute **A3**-action, sending the message $\langle \text{FINISH} \rangle$ around the ring.

Every process but L will receive the message $\langle \text{FINISH} \rangle$ and execute **A4**-action, which will be its final action. Finally L executes **A6**-action, ending the execution. So A_k solves

the process-terminating leader election for $\mathcal{A} \cap \mathcal{K}_k$. ■

Theorem 3.10

A_k has time complexity at most $(2k+2)n$, has message complexity at most $n^2(2k+1)$, and requires at most $(2k+1)nb + 2b + 3$ bits in each process.

Proof: Let $M = \max\{\text{mlty}(\ell) : \ell \text{ is a label in } \mathcal{R}\}$ and $m = \lceil (2k+1)/M \rceil\, n$. After at most m time units, L can determine that it is the true leader and send a message $\langle$FINISH$\rangle$. In n additional time units, $\langle$FINISH$\rangle$ traverses the whole ring and comes back to L to conclude the execution. In the worst case, there are no duplicate labels, *i.e.*, $M = 1$. Hence, the time complexity of A_k is at most $(2k+2)n$ time units.

When the execution halts, all sent messages have been received. So, the number of message sendings is equal to the number of message receptions. Each token initiated at the beginning of the growing phase circulates in the ring until being consumed by L after it realizes that it is the true leader. Similarly, $\langle$FINISH$\rangle$ traverses the ring once and stopped at L. Hence, each process receives at most as many messages as L. L receives $2k+1$ messages with the same label x to detect that it is the true leader (**A3**-action). When L becomes leader, the received token $\langle x \rangle$ is consumed and L has received messages containing other labels (at most $n-1$ different labels) at most $2k$ times each. Then, L receives and consumes all other tokens (at most $n-1$) before receiving $\langle$FINISH$\rangle$. Overall, L receives at most $n(2k+1)+1$ messages and so, the message complexity is at most $n^2(2k+1)+n$.

From the previous discussion, the length of $L.string$ is bounded by $2kn+1$. If $p \neq L$, then $p.string$ continues to grow after L executes **A3**-action until p executes **A4**-action by receiving the message $\langle$FINISH$\rangle$. Now, the FIFO property ensures that $p.string$ is appended at most $n-1$ times more than $L.string$ due to the remaining tokens. Thus the length of $p.string$ is always less than $(2k+1)n$. So, the space complexity is at most $(2k+1)nb + 2b + 3$ bits per process. ■

3.6 Algorithm B_k of Leader Election in $\mathcal{A} \cap \mathcal{K}_k$

For any $k \geq 1$, we now give another leader election algorithm, B_k, for a ring $\mathcal{R}$ in the class $\mathcal{A} \cap \mathcal{K}_k$. The space complexity of B_k is smaller than that of A_k, but its time complexity is greater. See Algorithm 3 for its code and Figure 3.6 for its state diagram.

3.6.1 Overview of B_k

Like A_k, B_k elects the true leader of $\mathcal{R}$, namely, the process L such that $LSeq(L)^n$ is a Lyndon word, *i.e.*, $LSeq(L)^n$ is minimum among the sequences $LSeq(q)^n$ of all processes q, where sequences are compared using lexicographical ordering.

The processes that are (still) competing to be the leader are said to be *active*. The other processes are said to be *passive*. Initially, the set of active processes contains all processes: $Act_0 = \{p_0, ..., p_{n-1}\}$. An execution of B_k consists of phases where processes are *deactivated*, *i.e.*, become passive. At the end of a given phase $i \geq 1$, the set of active processes is given by: $Act_i = \{p \in \mathcal{R} : LSeq(p)^i = LSeq(L)^i\}$, see Figure 3.7. During

Algorithm 3 – Actions of Process p in Algorithm B_k.

Inputs.

- $p.id \in \mathfrak{id}$

Variables.

- $p.state \in \{\textsc{Init}, \textsc{Compute}, \textsc{Passive}, \textsc{Shift}, \textsc{Win}, \textsc{Halt}\}$, initially $\textsc{Init}$
- $p.guest \in \mathfrak{id}$
- $p.inner \in \{1, \ldots, k\}$, initially 1
- $p.outer \in \{1, \ldots, k\}$, initially 1
- $p.leader \in \mathfrak{id}$
- $p.isLeader \in \mathbb{B} = \{\textsc{True}, \textsc{False}\}$, initially $\textsc{False}$
- $p.done \in \mathbb{B}$, initially $\textsc{False}$

Actions.

A1	::	$p.state = \textsc{Init}$	$\rightarrow$	$p.state := \textsc{Compute}$ $p.guest := p.id$ **send** $\langle p.guest \rangle$
(Computation During a Phase)				
A2	::	$p.state = \textsc{Compute} \wedge$ **rcv** $\langle x \rangle \wedge x > p.guest$	$\rightarrow$	(nothing)
A3	::	$p.state = \textsc{Compute} \wedge$ **rcv** $\langle x \rangle \wedge x = p.guest$ $\wedge\ p.inner < k$	$\rightarrow$	$p.inner{+}{+}$ **send** $\langle x \rangle$
A4	::	$p.state = \textsc{Compute} \wedge$ **rcv** $\langle x \rangle \wedge x < p.guest$	$\rightarrow$	$p.state := \textsc{Passive}$ **send** $\langle x \rangle$
(Phase Switching)				
A5	::	$p.state = \textsc{Compute} \wedge$ **rcv** $\langle x \rangle \wedge x = p.guest$ $\wedge\ p.inner = k$	$\rightarrow$	$p.state := \textsc{Shift}$ **send** $\langle \textsc{Phase\_Shift}, p.guest \rangle$
A6	::	$p.state = \textsc{Shift} \wedge$ **rcv** $\langle \textsc{Phase\_Shift}, x \rangle$ $\wedge\ (x \neq p.id \vee p.outer < k)$	$\rightarrow$	$p.state := \textsc{Compute}$ **if** $p.id = x$ **then** $p.outer{+}{+}$ $p.guest := x$ $p.inner := 1$ **send** $\langle p.guest \rangle$
(Passive Processes)				
A7	::	$p.state = \textsc{Passive} \wedge$ **rcv** $\langle x \rangle$	$\rightarrow$	**send** $\langle x \rangle$
A8	::	$p.state = \textsc{Passive} \wedge$ **rcv** $\langle \textsc{Phase\_Shift}, x \rangle$	$\rightarrow$	**send** $\langle \textsc{Phase\_Shift}, p.guest \rangle$ $p.guest := x$
(Ending Phase)				
A9	::	$p.state = \textsc{Shift} \wedge$ **rcv** $\langle \textsc{Phase\_Shift}, x \rangle$ $\wedge\ x = p.id \wedge p.outer = k$	$\rightarrow$	$p.state := \textsc{Win}$ $p.isLeader := \textsc{True}$ $p.leader := p.id$ $p.guest := p.id$ **send** $\langle \textsc{Finish}, p.id \rangle$
A10	::	$p.state = \textsc{Passive} \wedge$ **rcv** $\langle \textsc{Finish}, x \rangle$	$\rightarrow$	$p.state := \textsc{Halt}$ $p.leader := x$ $p.done := \textsc{True}$ **send** $\langle \textsc{Finish}, x \rangle$ (halt)
A11	::	$p.state = \textsc{Win} \wedge$ **rcv** $\langle \textsc{Finish}, x \rangle$	$\rightarrow$	$p.state := \textsc{Halt}$ $p.done := \textsc{True}$ (halt)

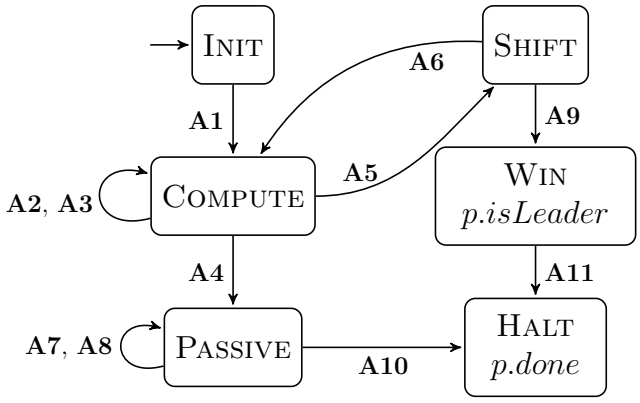
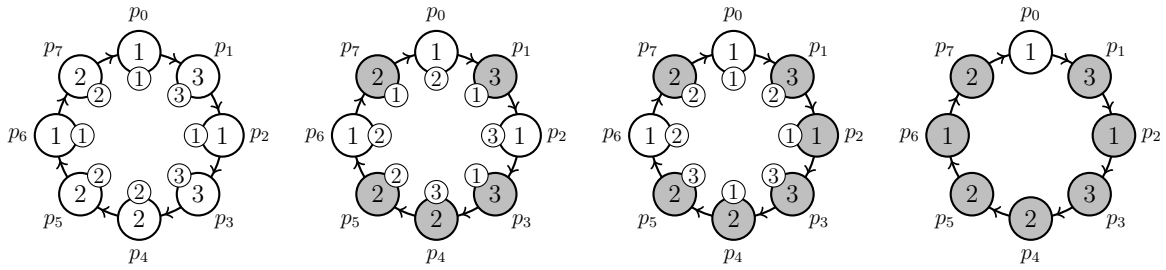


Figure 3.6 – State diagram of B_k.



(a) 1st phase. **(b)** 2nd phase. **(c)** 3rd phase. **(d)** 4th phase.

Figure 3.7 – Extracts from an example of execution of B_k where $k = 3$, showing the active (in white) and passive (in gray) processes at the beginning of each phase. The *guest* of a process is in the white bubble next to the corresponding node.

phase $i \geq 1$, a process q is removed from Act_i, when $LSeq(q)[i] > LSeq(L)[i]$; more precisely, when q realizes that some process $p \in Act_{i-1}$ satisfies $LSeq(p)[i] < LSeq(q)[i]$. When $i \geq n$, Act_i is reduced to $\{L\}$, since $\mathcal{R}$ is asymmetric. Using k, B_k is able to detect that at least n phases have been done, and so to terminate.

As defined in the specification, we use at each process p the constant $p.id$ and the variables $p.leader$ (of label type), $p.done$ and $p.isLeader$ (Booleans, initially False). Each process p also maintains a variable $p.state \in \{$Init, Compute, Shift, Passive, Win, Halt$\}$, initially equals to Init. A passive process is in state Passive; other states are used by (still) active processes; state Halt is the last state for every process.

Three kinds of message are exchanged: $\langle x \rangle$ is used during the computation of a phase, $\langle$Phase_Shift$, x\rangle$ is used to notify that a phase is over, and $\langle$Finish$, x\rangle$ is used during the ending phase, where x is of label type. Intuitively, we say that a process is in its i^{th} phase, with $i \geq 1$, if it received $(i-1)$ $\langle$Phase_Shift$, _\rangle$ messages.

Phase Computation. The goal of the i^{th} phase is to compute Act_i, given Act_{i-1}, namely to deactivate each active process p such that $LSeq(p)[i] > LSeq(L)[i]$. To that purpose, we introduce, at each process p, a variable $p.guest$, of label type, such that $p.guest = LSeq(p)[i]$. (How $p.guest$ is maintained in each phase will be explained later.)

During phase $i \geq 1$, the value $p.guest$ of every active process p circulates among active processes: at the beginning of the phase, every active process sends its current *guest* to its right neighbor (**A1**-action for the first phase, **A6**-action for other phases). Since passive processes are no more candidate, they simply forward the message (**A7**-action). When an active process p receives a label x greater than $p.guest$, it discards this value (**A2**-action), since $x > p.guest \geq LSeq(L)[i]$. Conversely, when p is active and receives a label x lower than $p.guest$, it turns to be passive, executing **A4**-action (nevertheless, p forwards x).

A process p, which is (still) active, can end the computation of its phase i once it has considered the *guest* value of every other process that are active all along phase i (*i.e.*, processes in Act_{i-1} that did not become passive during phase i). Such a process p detects the end of the current phase when it has seen the value $p.guest$ $(k+1)$ times. To that goal, we use the counter variable $p.inner$, which is initialized to 1 at the beginning of each phase (initialization and **A6**-action) and incremented each time p receives the value $p.guest$ while being active (**A3**-action) (once a process is passive the variable *inner* is meaningless). So, the current phase ends for an active process p when it receives $p.guest$ while $p.inner$ was already equal to k (**A5**-action).

Phase Switching. We now explain how $p.guest$ is maintained at each phase. Initially, $p.guest$ is set to $p.id$ and phase 1 starts for p (**A1**-action). Next, the value of $p.guest$ for every p is updated when switching to the next phase.

First, note that it is mandatory that every active process updates its *guest* variable when entering a new phase, *i.e.*, after detecting the end of the previous phase, so that the labels that circulate during the computation of the phase actually represent $LSeq(p)[i]$ for process $p \in Act_{i-1}$. Now, FIFO links allow to enforce a barrier synchronization as follows.

At the end of phase $i \geq 1$, Act_i is computed, and every still active process p has the same label prefix of length i, $LSeq(p)^i$, hence the same value for $p.guest = LSeq(p)[i]$. As a consequence, they are all able to detect the end of phase i. So, they switch their *state* from Compute to Shift and signal the end of the phase by sending a message $\langle$Phase_Shift, $p.guest\rangle$ (**A5**-action).

Messages $\langle$Phase_Shift, _$\rangle$ circulate in the ring, through passive processes (**A7**-action) until reaching another (or possibly the same) active process: when a process p (being passive or active) receives $\langle$Phase_Shift, $x\rangle$:

1. it switches from phase i to $(i+1)$ by adopting x as new *guest* value, and
2. if p is passive, it sends $\langle$Phase_Shift, $y\rangle$ where y was its previous *guest* value; otherwise, the shifting process is done and so p switches $p.state$ from Shift to Compute or Win and starts a new phase (**A6**-action or **A9**-action).

As a result, all *guest* values have eventually shifted by one process on the right for the next phase.

Note that, due to FIFO links and the fact that active processes switch to state Shift between two successive phases, phases cannot overlap, *i.e.*, when a label x is considered

in phase i, in state COMPUTE, x is the *guest* of some process q which is active in phase i, such that $LSeq(q)[i] = x$.

How Many Phases? Phase switching stops for an active process p once its *guest* took the value $p.id$ $(k+1)$ times. Indeed, when $p.guest$ is updated for the $(k+1)^{\text{th}}$ times by $p.id$, it is guaranteed that the number of phases executed by the algorithm is greater or equal to n, because $p.guest = LSeq(p)[i]$ and there is no more than k processes with the same value $p.id$. In this case, p is the true leader and every other process q is passive.

Again, to detect this, we use at each process p a counter called $p.outer$. It is initially set to 1 and incremented by each active process at each phase switching (**A6**-action). When $p.outer$ reaches the value $k+1$ (or equivalently when p receives $p.id$ while $p.outer = k$, see **A9**-action), p declares itself as the leader and initiates the final phase: it sends a message $\langle \text{FINISH}, p.id \rangle$; each other process successively receives the message, saves the label in the message in its *leader* variable, forwards the message, and then halts. Once the message reaches the leader (p) again, it also halts.

3.6.2 Correctness and Complexity Analysis

To prove the correctness of B_k (Theorem 3.11), we first establish that phases are causally well-defined (see Observation 3.1), *e.g.*, they do not overlap. Then, Lemmas 3.10-3.15 prove the invariant of the algorithm, by induction on the phase number. Finally, Theorem 3.12 proves its complexity.

First, a process p is in phase $i \geq 0$ if it set i times its variable $p.guest$. A barrier synchronization is achieved between each phase using messages $\langle \text{PHASE\_SHIFT}, \_ \rangle$. Hence we have the following observation:

Observation 3.1

Let $i \geq 1$. A message received in phase i has been sent in phase i (it was actually initiated in phase i). Conversely, if a message has been sent in phase i, it can only be received in phase i.

Proof: First, we prove some preliminary result.

Claim 1: Between two setting of $p.guest$, each process p sends and receives at least one message.

Proof of the claim: A process p can only set $p.guest$ executing **A1**, **A6**, **A8**, or **A9**-action. Furthermore, **A1**-action cannot be executed several times. Assume p sets $p.guest$ in step $\gamma_i \mapsto \gamma_{i+1}$ and then in step $\gamma_j \mapsto \gamma_{j+1}$. So, p executes **A6**, **A8**, or **A9**-action in $\gamma_j \mapsto \gamma_{j+1}$. Now, if p executes **A8**-action, it receives a message $\langle \text{PHASE\_SHIFT}, \_ \rangle$ and sends $\langle \text{PHASE\_SHIFT}, p.guest \rangle$ before updating $p.guest$ during step $\gamma_j \mapsto \gamma_{j+1}$. Similarly, if p executes **A6** or **A9**-action, it receives a message $\langle \text{PHASE\_SHIFT}, \_ \rangle$ in step $\gamma_j \mapsto \gamma_{j+1}$ before updating $p.guest$. Moreover, it necessarily executes **A5**-action beforehand, and so sends a message $\langle \text{PHASE\_SHIFT}, p.guest \rangle$. □

Then, assume by contradiction that some process q receives, in phase $j \geq 0$, a message m sent by its predecessor p in phase $i \geq 0$ such that $i \neq j$. Without loss of

generality, assume this is the first time a messsage is received in a phase different than the one of its sending.

Claim 2: $i \geq 1$ and $j \geq 1$

Proof of the claim: p cannot send messages before executing **A1**-action, *i.e.*, before setting $p.guest$ to $p.id$ and starting its first phase. Hence, $i \geq 1$. Similarly, q cannot receives any message before executing **A1**-action, so $j \geq 1$. □

Now, since m is the first problematic message and using Claim 1, we can deduce that $j = i - 1$ or $j = i + 1$. Let consider the two cases.

- If $j = i + 1$, then q updates its *guest* once more than p. Let consider the last time q updates its *guest* before receiving m, *i.e.*, the last time q executes **A1**, **A6**, **A8**, or **A9**-action to switch from its $(j-1)^{\text{th}}$ to its j^{th} phase. By claim 2, $i \geq 1$ so $j \geq 2$, and so, it does no execute **A1**-action to switch from its $(j-1)^{\text{th}}$ to its j^{th} phase, since **A1** can be executed only once.

  Now, if q executes **A6**, **A8**, or **A9**-action, it receives a messsage m' of the form $\langle \text{PHASE\_SHIFT}, \_ \rangle$ in phase $j-1$. Since m is the first problematic message, m' was sent by p in phase $j-1$. Either p executes **A8**-action or **A5**-action to send m'. In this latter case, p necessarily executes **A6**-action before sending a new message after m'. In both cases, p switches to phase j before sending m, a contradiction.
- If $j = i - 1$, then p updates its *guest* once more than q. Let consider the last time p updates its *guest* before sending m, *i.e.*, the last time p executes **A1**, **A6**, **A8**, or **A9**-action to switch from its $(i-1)^{\text{th}}$ to its i^{th} phase. Let consider each cases:
  - If p executes **A1**-action, then $i = 1$ since **A1**-action cannot be executed more than once. Now, by Claim 2, $j \geq 1$, a contradiction.
  - If p executes **A6** or **A9**-action, it necesserarily executes **A5**-action beforehand and so sends a message $m' = \langle \text{PHASE\_SHIFT}, \_ \rangle$ to q in phase $i-1$. Since m is the first problematic message, q receives m' in phase $i-1$ executing **A6**, **A8**, or **A9**-action. In every cases, q switches from phase $i-1$ to phase i before receiving m, a contradiction.
  - If p executes **A8**-action, it sends a message $m' = \langle \text{PHASE\_SHIFT}, \_ \rangle$ before switching to phase i. Again, since m is the first problematic message, q receives m' in phase $i-1$ executing **A6**, **A8**, or **A9**-action. In every cases, q switches from phase $i-1$ to phase i before receiving m, a contradiction. ■

In the following, we say that a process p is *deadlocked* if p is disabled although a message is ready to be received by p.

Definition 3.3 (HI_i)

Let $X = \min\{x : LSeq(L)^x \text{ contains } L.id(k+1) \text{ times}\}$. For any $i \in \{1, \ldots, X\}$, we define HI_i as the following predicate: $\forall p \in \mathcal{R}, \forall j, 1 \leq j < i$,

1. $p.guest$ is equal to $LSeq(p)[j]$ in phase j,
2. p is not deadlocked during its phase j, and
3. $p \in Act_j$ if and only if p exits its phase j using **A6** or **A9**-action.

Lemma 3.10

For all $i \in \{1, \ldots, X\}$, HI_i holds.

Lemma 3.10 is proven by induction on i. The base case ($i = 1$) is trivial. The induction step (assume HI_i and show HI_{i+1}, for $i \in \{1, \ldots, X-1\}$) consists in proving the correct behavior of phase i. To that goal, we prove Lemmas 3.11, 3.14, and 3.15 which respectively show Conditions 1, 2, and 3 for HI_{i+1}.

Lemma 3.11

For $i \in \{1, \ldots, X-1\}$, if HI_i holds, then $\forall p \in R, \forall j < i+1$, $p.guest$ is equal to $LSeq(p)[j]$ in phase j.

Proof: Let $i \in \{1, \ldots, X-1\}$ such that HI_i holds. First note that for every process p, we have $LSeq(p)[1] = p.id = p.guest$ in phase 1. Hence the lemma holds for $i = 1$. Now assume that $i > 1$. Using HI_i, we have that for every $1 \leq j < i$, $LSeq(p)[j] = p.guest$ at phase j.

We consider now the case when $j = i$. Note that a process can only change the value of its variable *guest* with **A6**, **A8** or **A9**-action, namely during phase switching. Let p be a process at phase i and consider, in the execution, the step when p switches from phase $(i-1)$ to phase i: it receives from its left neighbor, q a message $\langle \text{PHASE\_SHIFT}, x \rangle$, where x was the value of $q.guest$ when q sent the message (see **A5** and **A8**-actions). From Observation 3.1, and since p receives it at phase $(i-1)$, q sends this message at phase $(i-1)$ also. Hence, $x = q.guest$ at phase $(i-1)$. Now, when p receives the message, it assigns its variable $p.guest$ to x (**A6**, **A8**, or **A9**-action): hence, at phase i, $p.guest = LSeq(q)[i-1] = LSeq(p)[i]$. ■

From Observation 3.1, if p receives $\langle \text{PHASE\_SHIFT}, \_ \rangle$ at phase $i \geq 1$, it was sent by its left neighbor in phase i. So by Lemma 3.11, we deduce the following corollary.

Corollary 3.4

*For $i \in \{1, \ldots, X-1\}$, if HI_i holds, then $\forall p \in \mathcal{R}$, if p exits phase $j \leq i$ by **A9**-action, then $LSeq(p)[j]$ equals $p.id$.*

Lemma 3.12

*For $i \in \{1, \ldots, X-1\}$, if HI_i holds, then no **A9**-action is executed before phase $i+1$.*

Proof: Assume by contradiction that HI_i holds and some **A9**-action is executed before phase $i+1$. Consider the first time it occurs, say some process p executes **A9**-action in some phase $j \leq i$.

From Corollary 3.4, by **A9**-action, p receives a message $\langle \text{PHASE\_SHIFT}, x \rangle$ with $x = p.id = LSeq(p)[j]$.

Furthermore, we have that $p.outer = k$ at phase j. Hence $p.id$ was observed $(k+1)$ times since the beginning of the execution: $p.guest$ took k times value $p.id$ and the value x in the received message is also $p.id$. By Lemma 3.11, the sequence of values of $p.guest$

is equal to $LSeq(p)^{j-1}$. Adding $x = LSeq(p)[j]$ at the end of the sequence, we obtain $LSeq(p)^j$. Hence, $j = \min\{x : LSeq(p)^x$ contains $p.id$ $(k+1)$ times$\}$ and $n < j$ (this implies that $j \geq 2$, hence $(j-1) \geq 1$).

As p executes **A9**-action in phase j, it is active during its whole j^{th} phase and hence exits its phase $(j-1)$ using **A6**-action. By Condition 3 in HI_i and since $(j-1) < i$, $p \in Act_{j-1}$. By definition of Act_{j-1}, since $j > n$, $Act_{j-1} = \{L\}$, hence $p = L$.

As a consequence, $j = X$, a contradiction. ■

In the following, we show that processes cannot deadlock (Lemma 3.14). We start by showing the following intermediate result:

Lemma 3.13

While a process is in state Compute *(resp.* Shift*), the next message it has to consider cannot be of the form* $\langle$Phase_Shift, _$\rangle$ *(resp.* $\langle x \rangle$*).*

Proof: Assume by contradiction that some process p is in state Compute (resp. Shift), but receives an unexpected message $\langle$Phase_Shift, _$\rangle$ (resp. $\langle x \rangle$) meanwhile. We examine the first case, the other case being similar. The unexpected message was transmitted through passive processes to p, but first initiated by some active process q (**A5**-action).

Since **A5**-action was enabled at process q, q received k messages $\langle q.guest \rangle$ during one and the same phase. By the multiplicity, at least one of those messages, say m, was initiated by q using **A1** or **A6**-action. So, m traversed the entire ring (**A2**-**A5**, **A7**-actions). Observation 3.1 ensures that this traversal occurs during one and the same phase. As a consequence, $q.guest \geq r.guest$ for every process r that were active when receiving m. In particular, $q.guest \geq p.guest$.

As q executed **A5**-action, k messages $\langle q.guest \rangle$ were sent by q (one action, either **A1** or **A6**-action, and $(k-1)$ **A3**-actions) during the traversal of m, and so during the same phase again. Hence, p has also received $\langle q.guest \rangle$ k times during the same phase. Thus, $p.guest \geq q.guest$ since p is still active, and so $p.guest = q.guest$. Now, counters *inner* of p and q counted accordingly during this phase: $p.inner$ should be greater than or equal to k. Hence p should have executed **A5**-action before receiving the unexpected message, a contradiction. ■

Lemma 3.14

For every $i \in \{1, \ldots, X-1\}$*, if* HI_i *holds, then* $\forall p \in \mathcal{R}$*,* p *is not deadlocked before phase* $(i+1)$*.*

Proof: Let $i \in \{1, \ldots, X-1\}$ such that HI_i holds. Let p be any process. If p is in state Init or Passive in phase i, then it cannot deadlock since the states Init and Passive are not blocking by definition of the algorithm. From Lemma 3.12 since HI_i holds, p cannot take state Win before phase $(i+1)$. Hence, it cannot take state Halt by **A11**-action. As no **A9**-action is executed during phase i, no message $\langle$Finish, _$\rangle$ circulates in the ring during this phase (Observation 3.1): **A10**-action cannot be enabled, hence p cannot take state Halt by **A10**-action as well. If p is in state Compute (resp. Shift), it cannot receive any message $\langle$Phase_Shift, _$\rangle$ (resp. $\langle x \rangle$) by Lemma 3.13. Moreover,

it cannot have received any message $\langle \text{FINISH}, \_ \rangle$ since no such message was sent during this phase (see Lemma 3.12 which applies as HI_i holds). As a conclusion, there is no way for p to deadlock during phase i. ■

Lemma 3.15

For every $i \in \{1, \ldots, X-1\}$, if HI_i holds, then $\forall p \in \mathcal{R}$, $\forall j < i+1$, $p \in Act_j$ if and only if p exits its phase j by **A6** *or* **A9***-action.*

Proof: Let $i \in \{1, \ldots, X-1\}$ such that HI_i holds.

Claim 1: $\forall p$, if $p \in Act_{i-1}$ (resp. $\notin Act_{i-1}$), p initiates (resp. does not initiate) a message $\langle LSeq(p)[i] \rangle$ (resp. any message) at the beginning of phase i.

Proof of the claim: If $i = 1$, every process p is in Act_0 and starts its phase 1, *i.e.*, its execution, by executing **A1**-action and sending its label $p.id = LSeq(p)[1]$. Otherwise ($i > 1$), by Lemma 3.12, no process can execute **A9**-action before phase $(i+1)$. So by HI_i, every process $p \in Act_{i-1}$ exits phase $(i-1)$ (and so starts phase i) by executing **A6**-action and sending its label $p.guest = LSeq(p)[i]$ (Lemma 3.11). By HI_i, if p is not in Act_{i-1}, p does not exits phase $(i-1)$ by executing **A6**-action and so it cannot initiates a message with its label at the beginning of phase i. □

Claim 2: Any process p receives a message $\langle LSeq(L)[i] \rangle$ k times during its phase i.

Proof of the claim: Consider a message $m = \langle LSeq(L)[i] \rangle$ that circulates the ring (at least one is circulating since $L \in Act_{i-1}$ initiates one at the beginning of phase i, see Claim 1). m is always received in phase i (see Observation 3.1) all along its ring traversal. From HI_i and Lemma 3.14, no process is deadlocked before its phase $(i+1)$. Hence, when m reaches a process in state PASSIVE, it is forwarded (**A7**-action) and when m reaches a process q in state COMPUTE (with $q.guest = LSeq(q)[i] \geq LSeq(L)[i]$, by Lemma 3.11 and definition of L), it is also forwarded unless **A5**-action is enabled at q. This occurs at q if $LSeq(q)[i] = LSeq(L)[i]$, since $q.inner$ is initialized to 1 at the beginning of the phase (**A1** or **A6**-action) and incremented if q receives $LSeq(q)[i]$. Hence, q has received k messages $\langle LSeq(L)[i] \rangle$ during the phase.

As a consequence, between any two processes q and q' in Act_{i-1} (in state COMPUTE in phase i, see HI_i) such that $LSeq(q)[i] = LSeq(q')[i] = LSeq(L)[i]$, k messages $\langle LSeq(L)[i] \rangle$ circulates during phase i; any process between q and q' has forwarded them (and so received them). □

By HI_i, the lemma holds for all $j < i$. Let now consider the case $j = i$.

If $p \in Act_i$, then $LSeq(p)^i = LSeq(L)^i$ and in particular, $LSeq(p)[i] = LSeq(L)[i]$. As $Act_i \subseteq Act_{i-1}$, p is active at the end of phase $(i-1)$ and as no **A9**-action can take place before phase $(i+1)$ (Lemma 3.12), p is in state COMPUTE during the computation of phase i. Since $p.guest = LSeq(L)[i] \leq LSeq(q)[i]$ for any $q \in Act_{i-1}$ (Lemma 3.11, definition of L), and as any message $\langle x \rangle$ that circulates during the phase is initiated by some process $q \in Act_{i-1}$ with $x = LSeq(q)[i]$ (HI_i and Claim 1), p never executes **A4**-action during phase i. Furthermore, p receives k times $p.guest$ during the phase (Claim 2), hence it executes **A5**-action followed by **A6** or **A9**-action to exit phase i.

Conversely, if $p \notin Act_i$, it may be or not in Act_{i-1}. If $p \notin Act_{i-1}$, then from HI_i, p exits phase $(i-1)$ with **A8**-action; it remains in state PASSIVE all along phase i and can only exit phase i with **A8**-action. Otherwise, $p \in Act_{i-1}$, *i.e.*, $LSeq(p)^{i-1} = LSeq(L)^{i-1}$ but $LSeq(p)[i] > LSeq(L)[i]$. p executes **A4**-action at least when receiving the first occurrence of $\langle LSeq(L)[i] \rangle$ (Claim 2) and takes state PASSIVE. Once p is passive, it remains so and can only exit phase i using **A8**-action.

Finally, at least L executes **A5**-action: hence phase switching actually occurs (started by L or some other process) and causes every process to exit phase i. ■

This ends the proof of Lemma 3.10.

Theorem 3.11

B_k solves the process-terminating leader election for $\mathcal{A} \cap \mathcal{K}_k$.

Proof: By Lemma 3.10 and by definition of X, no process is deadlocked before phase X and L is the only process that exits phase X executing **A6** or **A9**-action. Now, by Lemma 3.10 and Corollary 3.4, $\forall i \in \{1, \ldots, X\}$, $L.guest = LSeq(L)[i]$ during phase i. Hence, when p begins its X^{th} phase, it is the $(k+1)^{\text{th}}$ time that L sets $L.guest$ to $L.id$. Since $L.outer$ is initialized to 1 and incremented when L enters a new phase with $L.guest = L.id$, L enters its phase X by **A9**-action. So, L sends a message $\langle \text{FINISH}, L.id \rangle$. L also sets $L.isLeader$ and $L.leader$ to TRUE and $L.id$, respectively. Every other process p receives the message in phase X (Observation 3.1) while being in state PASSIVE, since p exits its $(X-1)^{\text{th}}$ phase executing **A8**-action (Lemma 3.10). So, p saves $L.id$ in its variable *leader*, then transmits the message to its right neighbor, and finally halts (**A10**-action). Finally, L receives $\langle \text{FINISH}, L.id \rangle$ and halts (**A11**-action). ■

Theorem 3.12

B_k has time complexity $O(k^2n^2)$, message complexity $O(k^2n^2)$, and requires $2\lceil \log k \rceil + 3b + 5$ bits per process.

Proof: A phase ends when an active process sees its *guest* $(k+1)$ times. This requires $O((k+1)n)$ time units. There is exactly X phases and $X \leq (k+1)n$. Thus, the time complexity of B_k is $O(k^2n^2)$.

During the first phase, every process starts by sending its *id*. Since a phase involves $O((k+1)n)$ actions per process, each process forwards labels $O((k+1)n)$ times. Finally, to end the first phase, every process sends and receives $\langle \text{PHASE\_SHIFT}, \_ \rangle$. Hence, $O(kn^2)$ messages are sent during the first phase. Moreover, only processes that have the same label as L (at most k) are still active after the first phase.

For every phase $i > 1$, let $d = \text{mlty}(\min\{p.guest : p \in Act_{i-1}\})$. When phase i starts, every active process (at most k) sends its new *guest*. When the first message ends its first traversal ($O(kn)$ messages), every process that becomes passive in the phase is already passive. Then, the variables *inner* of the remaining active processes increment of d each turn of ring by a message. So the remaining messages (at most d) do at most $\frac{k}{d}$ traversal (n hops): $O(kn)$ messages. Overall, the phase requires $O(kn)$ messages exchanged. As there is at most $O(kn)$ phases, there are at most $O(k^2n^2)$ messages exchanged.

Class		Proved in
Symmetrical	Message-terminating leader election impossible	Theo. 3.1
$\mathcal{K}_k$	Message-terminating leader election impossible	Theo. 3.2
$\mathcal{U}^*$	Process-terminating leader election impossible	Theo. 3.3
$\mathcal{A}$	Process-terminating leader election impossible	Theo. 3.4

Class	Lower Bound on Time	Algo.	Time	Nbr of Msgs	Memory
$\mathcal{U}^* \cap \mathcal{K}_k$	$\Omega(kn)$ (Cor. 3.1)	U_k	$n(k+2)$	$O(n^2+kn)$	$\lceil \log(k+1) \rceil + 2b + 4$
$\mathcal{A} \cap \mathcal{K}_k$	$\Omega(kn)$ (Cor. 3.2)	A_k	$(2k+2)n$	$n^2(2k+1)$	$2(k+1)nb + 2b + 3$
		B_k	$O(k^2n^2)$	$O(k^2n^2)$	$2\lceil \log k \rceil + 3b + 5$

Table 3.1 – Summary of Chapter 3 results.

Finally, for every process p, $p.inner$ and $p.outer$ are initialized to 1 and they are never incremented over k. Hence, every process requires $2\lceil \log k \rceil + 3b + 5$ bits. ■

3.7 Conclusion

Summary of Contributions. In this chapter, we have studied the leader election problem in unidirectional ring networks with homonym processes. The whole results are sum up in Table 3.1.

We have proven that message-terminating leader election is impossible to solve in unidirectional ring networks with a symmetrical labelling and in the class $\mathcal{K}_k$, $k \geq 2$, of unidirectionnal rings where no more than k processes share the same label. We have also proven that process-terminating leader election is impossible to solve in the class $\mathcal{U}^*$ of unidirectionnal ring networks containing at least one process with a unique label. This result naturally extends to the class $\mathcal{A}$ of unidirectionnal ring networks with an asymmetrical labelling.

Then, we have proposed three algorithms. Algorithm U_k solves process-terminating leader election for class $\mathcal{U}^* \cap \mathcal{K}_k$, for any $k \geq 1$, in $n(k+2)$ time units. Its message complexity is $O(n^2+kn)$ and it requires $\lceil \log(k+1) \rceil + 2b + 4$ bits per process, where b is the number of bits required to store a label. U_k is asymptotically optimal in time and memory.

Algorithms A_k and B_k both solves process-terminating leader election for class $\mathcal{A} \cap \mathcal{K}_k$, for any $k \geq 1$. A_k is asymptotically optimal in time, with at most $(2k+2)n$ time units, but it requires $2(k+1)nb + 2b + 3$ bits per process and at most $n^2(2k+1)$ messages are exchanged during an execution. On the contrary, B_k is asymptotically optimal in memory since it requires only $2\lceil \log k \rceil + 3b + 5$ bits per process, but its time complexity is $O(k^2n^2)$ and its message complexity is $O(k^2n^2)$.

Perspectives. First, the amount of bits exchanged in an execution of U_k is very closed to the lower bound we proved ($\Omega(kn+n^2)$) with $O((kn+n^2)b)$ bits exchanged, where b is the number of bits required to store a label. Notice that $b = \lceil \log n \rceil$ if we consider that labels are natural integers like commonly done in the litterature.

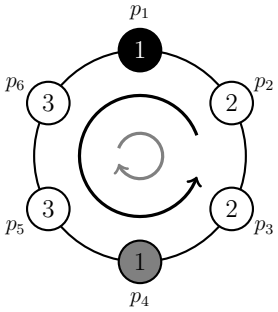


Figure 3.8 – Processes elected by both instances of A_k of B_k on bidirectionnal ring. In gray, the leader designated by the instance running on the clockwise orientation. In black, the leader designated by the instance runnning on the anticlockwise orientation.

On the contrary, the amount of bits exchanged in an execution of A_k or B_k is greater, respectively $O(n^2(2k+1)b)$ and $O(k^2n^2b)$ bits exchanged. Whether it is possible to reduce the amount of exchanged information without degrading the other performances of the algorithms (time complexity for A_k, memory requirement for B_k) is worth investigating.

Furthermore, we can easily transform Algorithm U_k to solve process-terminating leader election in bidirectionnal ring networks where at least one process has a unique label, even if processes do not share a common sense of direction. Indeed, we can execute two instances of U_k on each process, one for each orientation. More precisely, each process executes an instance of U_k managing messages coming from its (local) left neighbor and another instance managing messages coming from its (local) right neighbor. When a process receives a message from a neighbor, it sends a message, if needed, to its other neighbor. Since the rule to choose the leader does not depend on the orientation of the ring (*i.e.*, the process with the smallest unique label), both instances of the algorithm designate the same process and the elected process can declare itself leader.

Nonetheless, we cannot generalize Algorithms A_k and B_k with this scheme. Indeed, the rule to choose the leader in A_k and B_k depends on the orientation of the ring, and so the two instances may not designate the same process as leader. For example, on Figure 3.8, the leader chosen by the clockwise orientation is p_4 while the leader chosen by the anticlockwise orientation is p_1. Further work is then needed to solve process-terminating leader election in bidirectionnal rings that do not contain a unique process.

Chapter 4

Self-stabilizing Leader Election under Unfair Daemon

"Of all the trees we could've hit, we had to get one that hits back."
— J.K. Rowling, *Harry Potter and the Chamber of Secrets*

Contents

4.1 Introduction

Similarly to Chapter 3, we consider here the problem of leader election, *i.e.*, we want to distinguish a unique process, so called leader, and every process eventually knows the leader ID. But, contrary to Chapter 3, we assume fully identified networks.

We aim to design (deterministic) silent self-stabilizing leader election for connected identified networks of arbitrary topology. In the locally shared memory model, silent

means that executions are finite [DGS99].

4.1.1 Related Work

Leader election problem in general and self-stabilizing leader election have been extensively studied. We focus here on self-stabilizing solutions for arbitrary network topologies.

In [DGS99], Dolev et al. showed that silent self-stabilizing leader election requires $\Omega(\log n)$ bits per process, where n is the number of processes. Notice that non-silent self-stabilizing leader election algorithm can be achieved while using less memory, see for example the non-silent self-stabilizing leader election algorithm for unoriented ring networks [BT17a] (respectively, for arbitrary networks [BT17b]) of Blin and Tixeui that only requires $O(\log \log n)$ space (respectively, $O(\max(\log \Delta, \log \log n))$, where Δ is the degree of the network) per process.

Some self-stabilizing leader election algorithms for arbitrary connected identified networks have been proposed in the message-passing model [ABB98, AKM+93, BK07].

First, in [ABB98], Afek and Bremler propose an algorithm stabilizing in $O(n)$ rounds using $\Theta(\log n)$ bits per process. But it assumes that the link-capacity, *i.e.*, the amount of information that can circulate in the link at any moment, is bounded by a value B, known by every process.

Two different solutions that stabilizes in $O(\mathcal{D})$ rounds, where $\mathcal{D}$ is the diameter of the network, were proposed in [AKM+93, BK07]. However, both solutions assume that the processes know an upper bound D on the diameter $\mathcal{D}$ and they requires $\Theta(\log D \log n)$ bits per process.

Several solutions were also proposed in the locally shared memory model [DH97, AG94, DLP10, DLV11a, DLV11b, KK13].

In [DH97], Dolev and Herman propose a non-silent algorithm working under a strongly fair daemon. They assume that every process knows an upper bound N on the number of processes. This solution stabilizes in $O(\mathcal{D})$ rounds using $\Theta(N \log N)$ bits per process.

The algorithm proposed by Arora and Gouda in [AG94] works under a weakly fair daemon and also assumes an upper bound N on the number of processes. This solution stabilizes in $O(N)$ rounds and requires $\Theta(N \log N)$ bits per process.

The solution of Datta et al. in [DLP10] is the first self-stabilizing leader election algorithm for arbitrary connected identified networks that is proved under the distributed unfair daemon. This algorithm stabilizes in $O(\mathcal{D})$ rounds. However, the space complexity is unbounded. More precisely, the algorithm requires for each process to maintain an unbounded integer in its local memory.

The three other solutions [DLV11a, DLV11b, KK13] are all asymptotically optimal in memory, *i.e.*, they requires $\Theta(\log n)$ bits per process.

In [KK13], Kravchik and Kutten propose an algorithm working under the synchronous daemon. The stabilization time of this latter algorithm is in $O(\mathcal{D})$ rounds.

Finally, the two solutions proposed by Datta et al. in [DLV11a, DLV11b] assume a

distributed unfair daemon and have a stabilization time in $O(n)$ rounds. However, even if these two algorithms stabilizes within a finite number of steps, since they are proved under an unfair daemon, no step complexity is given.

4.1.2 Contributions

In this chapter, we study the silent self-stabilizing leader election problem in arbitrary static connected and identified networks. Our solution, denoted $\mathcal{LE}$, is written in the locally shared memory model and assumes a distributed unfair daemon, the weakest scheduling assumption. It assumes no knowledge of any global parameter (*e.g.*, an upper bound on $\mathcal{D}$ or n) of the network.

This solution is presented and proved in Section 4.3. Like previous solutions of the literature [DLV11a, DLV11b], it stabilizes in $\Theta(n)$ rounds in the worst case and it is asymptotically optimal in space. Indeed, it requires $\Theta(\log n + b)$ bits per process, where b is the number of bits required to store an ID. If we consider that IDs are natural integers as it is commonly done in the literature, $b = \log n$ and so $\mathcal{LE}$ can be implemented using $\Theta(\log n)$ bits per process. Yet, contrary to [DLV11a, DLV11b], we show that our algorithm has a stabilization time in $\Theta(n^3)$ steps in the worst case.

For fair comparison, we also studied the step complexity of the algorithms given in [DLV11a, DLV11b], noted here $\mathcal{DLV}_1$ (see Section 4.4) and $\mathcal{DLV}_2$ (see Section 4.5), respectively. These latter are the closest to ours in terms of assumptions and performance. We show that their stabilization time is not polynomial.

Indeed, for $n \geq 5$, there exists a network of n processes and a possible execution of $\mathcal{DLV}_1$ that stabilizes in $\Omega(2^{\lfloor \frac{n-1}{4} \rfloor})$ steps.

Similarly, there is no constant α such that the stabilization time of $\mathcal{DLV}_2$ is in $O(n^\alpha)$ steps. More precisely, we show that fixing α to any constant greater than or equal to 4, for every $\beta \geq 2$, there exists a network of $n = 2^{\alpha-1} \times \beta$ processes in which there exists a possible execution that stabilizes in $\Omega(n^\alpha)$ steps.

These results were published in the proceedings of the 16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2014) [ACD+14], in the special issue of SSS 2014 in Information and Computation [ACD+16], and in the proceedings of the 17èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications (ALGOTEL 2015) [ACD+15].

4.2 Preliminaries

In this section, we detail the context (Section 4.2.1) and we define the considered leader election problem (Section 4.2.2).

4.2.1 Context

We consider static bidirectional and identified networks of arbitrary connected topology. We assume the locally shared memory model presented in Section 2.6 under the

distributed unfair daemon. We denote by $n \geq 1$ the number of processes and $\mathcal{D}$ the diameter of the network.

We also denote by ℓ the process of minimum ID. By abuse of notation, we identify a process with its ID in the explanations, whenever convenient.

4.2.2 Silent Self-stabilizing Leader Election

We define the specification of the leader election problem, denoted SP_{LE}. We denote $Leader : V \to \mathfrak{id}$ the function defined on the state of any process $p \in V$ that returns the ID of the leader designed by p.

Definition 4.1 (*Leader Election*)

An algorithm Alg solves the *leader election* problem if any execution $(\gamma_i)_{i\geq 0} \in \mathcal{E}_{\text{Alg}}$ satisfies the following conditions:

1. For each configuration γ_i, $i \geq 0$, for every pair of processes $p, q \in V$, $Leader(p) = Leader(q)$ in γ_i and $Leader(p)$ is the ID of an (existing) process.
2. For each configuration γ_i, $i \geq 1$, for every process p, $Leader(p)$ has the same value in γ_i and in γ_0.

In this chapter, we aim to design a self-stabilizing and silent leader election algorithm. An algorithm is *silent* if all its execution are finite. Hence, to prove that a leader election algorithm is self-stabilizing and silent, it is necessary and sufficient to show that:

1. Every execution is finite.
2. In every terminal configuration, for every pair of processes $p, q \in V$, $Leader(p) = Leader(q)$ and $Leader(p)$ is the ID of some process.

4.3 Algorithm $\mathcal{LE}$

In this section, we present a silent and self-stabilizing leader election algorithm, called $\mathcal{LE}$. Its formal code is given in Algorithm 4.

4.3.1 Overview of $\mathcal{LE}$

Starting from an arbitrary configuration, $\mathcal{LE}$ converges to a terminal configuration where the process of minimum ID, ℓ, is elected. More precisely, in the terminal configuration, every process p knows the identifier of ℓ thanks to its local variable $p.idRoot$; moreover a spanning tree rooted at ℓ is defined using two variables per process: *par* and *level*. Formally:

1. $\ell.idRoot = \ell.id$, $\ell.par = \ell$, and $\ell.level = 0$, and
2. $\forall p \neq \ell.id$, $p.idRoot = \ell$, $p.par$ points to the parent of p in the tree and $p.level$ is the level of p in the tree.

Non Self-stabilizing Leader Election. We first consider a simplified version of $\mathcal{LE}$. Starting from a predefined initial configuration, it elects ℓ in all *idRoot* variables and builds a spanning tree rooted at ℓ.

Initially, every process p declares itself as leader: $p.idRoot = p.id$, $p.par = p$, and $p.level = 0$. So, p satisfies the two following predicates:

$$\begin{array}{lcl} SelfRoot(p) & \equiv & p.par = p \\ SelfRootOk'(p) & \equiv & (p.level = 0) \wedge (p.idRoot = p) \end{array}$$

Note that, in the sequel, we say that p is a *self root* when $SelfRoot(p)$ holds.

From such an initial configuration, our non self-stabilizing algorithm consists in the following single action:

$$\begin{array}{lcll} \textbf{J'} & :: & \exists q \in p.\mathcal{N}, (q.idRoot < p.idRoot) & \rightarrow \quad p.par := \min_{\preceq}\{q \in p.\mathcal{N}\} \\ & & & \quad\;\; p.idRoot := p.par.idRoot \\ & & & \quad\;\; p.level := p.par.level + 1 \end{array}$$

where $\forall x, y \in V, x \preceq y \Leftrightarrow (x.idRoot \leq y.idRoot) \wedge [(x.idRoot = y.idRoot) \Rightarrow (x.id < y.id)]$.

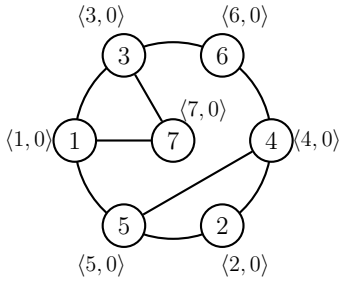
Informally, when p discovers that $p.idRoot$ is not equal to the minimum identifier, it updates its variables accordingly: let q be the neighbor of p having $idRoot$minimum. Then, p selects q as new parent (*i.e.*, $p.par := q$ and $p.level := p.par.level + 1$) and sets $p.idRoot$ to the value of $q.idRoot$. If there are several neighbors having minimum $idRoot$, we break ties using the identifiers of those neighbors.

Hence, the identifier of ℓ is propagated, from neighbors to neighbors, into the $idRoot$ variables and the system reaches a terminal configuration in $O(\mathcal{D})$ rounds. Figure 4.1 shows an example of such an execution.

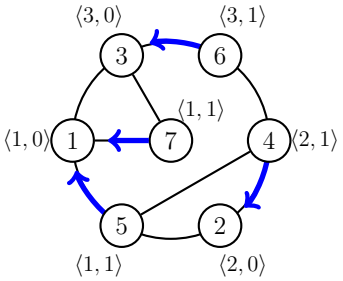
Notice first that for every process p, $p.idRoot$ is always less than or equal to its own identifier. Indeed, $p.idRoot$ is initialized to p and decreases each time p executes **J'**-action. Hence, $p.idRoot = p$ while p is a self root and after p executes **J'**-action for the first time, $p.idRoot$ is smaller than its ID forever.

Second, even in this simplified context, for each two neighbors p and q such that q is the parent of p, it may happens that $p.idRoot$ is greater than $q.idRoot$ — an example is shown in Figure 4.1c, where $p.id = 6$ and $q.id = 3$. This is due to the fact that p joins the tree of q but meanwhile q joins another tree and this change is not yet propagated to p. Similarly, when $p.idRoot \neq q.idRoot$, $p.level$ may be different from $q.level+1$. According to those remarks, we can deduce that when $p.par = q$ with $q \neq p$, we have the following relation between p and q:

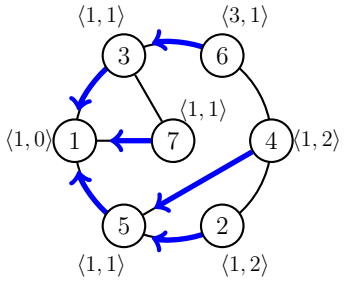
$$\begin{array}{lcl} GoodIdRoot(p,q) & \equiv & (p.idRoot \geq q.idRoot) \wedge (p.idRoot < p.id) \\ GoodLevel(p,q) & \equiv & (p.idRoot = q.idRoot) \Rightarrow (p.level = q.level + 1) \end{array}$$



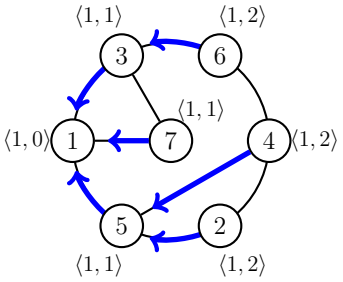
(a) Initial configuration. $SelfRoot(p) \wedge SelfRootOk'(p)$ holds for every process p.



(b) 4, 5, 6, and 7 executed **J'**-action. Note that **J'**-action was not enabled at 2 because it is a local minimum.

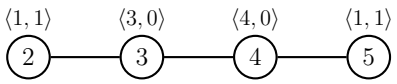


(c) 2, 3, and 4 executed **J'**-action. 3 joins the tree rooted at 1. However, the new value of 3.*idRoot* is not yet propagated to its child 6.

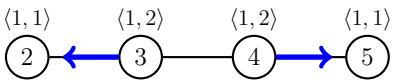


(d) 6 executed **J'**-action. The configuration is now terminal, $\ell = 1$ is elected, and a tree rooted at ℓ is available.

Figure 4.1 – Example of execution of the non self-stabilizing leader election algorithm. Process IDs are given inside the nodes. $\langle x, y \rangle$ means that $idRoot = x$ and $level = y$. Arrows represent *par* pointers. The absence of arrow means that the process is a self root.



(a) Illegitimate initial configuration, where 2 and 5 have fake *idRoot*.



(b) 3 and 4 executed **J'**-action. The configuration is terminal.

Figure 4.2 – Example of execution that does not converge to a legitimate configuration.

Fake IDs. This previous algorithm is not self-stabilizing. Indeed, in a self-stabilization context, the execution may start in an arbitrary configuration. In particular, *idRoot* variables can be initialized to arbitrary ID type values, even values that are actually not IDs of (existing) processes. We call such values *fake IDs*.

The existence of fake IDs may lead the system to an illegitimate terminal configuration. Refer to the example of execution given in Figure 4.2: starting from the configuration in 4.2a, if processes 3 and 4 move, the system reaches the terminal configuration given in 4.2b, where there are two trees and the *idRoot* variables elect the fake ID 1.

In this example, 2 and 5 can detect the problem. Indeed, predicate $SelfRootOk'$

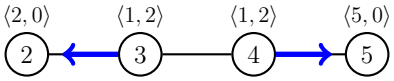


Figure 4.3 – One step after Figure 4.2b, 2 and 5 have reset.

is violated by both 2 and 5. One may believe that it is sufficient to reset the local state of processes which detect inconsistency (here processes 2 and 5) to $p.idRoot := p.id$, $p.par := p$ and $p.level := 0$. After these resets, there are still some errors, as shown on Figure 4.3. Again, 3 and 4 can detect the problem. Indeed, predicate $GoodIdRoot(p, p.par) \wedge GoodLevel(p, p.par)$ is violated by both 3 and 4. In this example, after 3 and 4 have reset, all inconsistencies have been removed. So let define the following action:

$$\begin{array}{lcll} \textbf{R'} & :: & \big(SelfRoot(p) \wedge \neg SelfRootOk'(p)\big) & \rightarrow \quad p.par := p \\ & & \quad \vee \big(GoodIdRoot(p, p.par) \wedge GoodLevel(p, p.par)\big) & \quad\quad p.idRoot := p.id \\ & & & \quad\quad p.level := 0 \end{array}$$

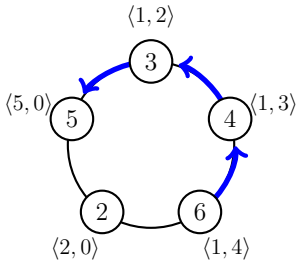
Unfortunately, this additional action does not ensure the convergence in all cases, see the example in Figure 4.4. Indeed, if a process resets, it becomes a self root but this does not erase the fake ID in the rest of its subtree. Then, another process can join the tree and adopt the fake ID which will be further propagated, and so on. In the example, a process resets while another joins its tree at lower level, and this leads to endless erroneous behavior, since we do not want to assume any maximal value for *level* (such an assumption would otherwise imply the knowledge of some upper bound on n). Therefore, the whole tree must be reset, instead of its root only. To that goal, we first freeze the "abnormal" tree in order to forbid any process to join it, then the tree is reset top-down. The cleaning mechanism is detailed in the next paragraph.

Abnormal Trees. To introduce the trees, we define what is a "good relation" between a parent and its children. Namely, the predicate $KinshipOk'(p, q)$ models that a process p is a *real child* of its parent $q = p.par$. This predicate holds if and only if $GoodLevel(p, q)$ and $GoodIdRoot(p, q)$ are TRUE. This relation defines a spanning forest: a *tree* is a maximal set of processes connected by *par* pointers and satisfying $KinshipOk'$ relation.

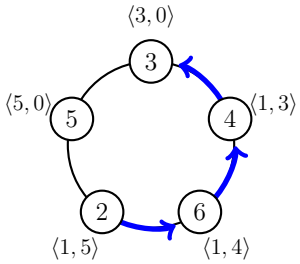
A process p is a root of such a tree whenever $SelfRoot(p)$ holds or $KinshipOk'(p, p.par)$ is FALSE. When $SelfRoot(p) \wedge SelfRootOk'(p)$ is TRUE, p is a *normal root* just as in the non self-stabilizing case. In other cases, there is an inconsistency and p is said to be an *abnormal root*:

$$\begin{array}{lcl} AbnormRoot'(p) & \equiv & \big(SelfRoot(p) \wedge \neg SelfRootOk'(p)\big) \\ & & \quad \vee \big(\neg SelfRoot(p) \wedge \neg KinshipOk'(p, p.par)\big) \end{array}$$

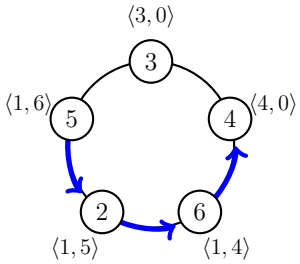
These are the two possible errors identified in the non self-stabilizing algorithm. A tree is called an *abnormal tree* (respectively *normal*) when its root is abnormal (respectively normal).



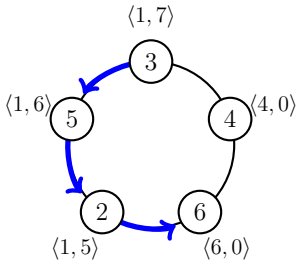
(a) Illegitimate initial configuration.



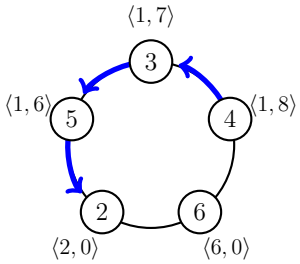
(b) 2 joins the tree. 3 leaves it.



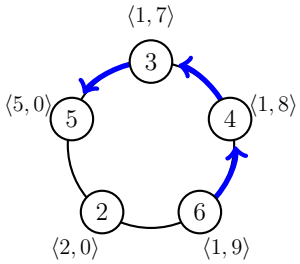
(c) 5 joins, 4 leaves.



(d) 3 joins, 6 leaves.



(e) 4 joins, 2 leaves.



(f) 6 joins, 5 leaves. Configuration similar to (a).

Figure 4.4 – The first process of the chain of arrows violates the predicate $SelfRootOk$ and resets by executing **R'**-action, while another process joins its tree. This cycle of resets and joins might never terminate.

We now detail the different variables, predicates, and actions of Algorithm 4.

Variable *status*. Abnormal trees need to be frozen before being cleaned in order to prevent them from growing endlessly. This mechanism (inspired from [BCV03]) is achieved using an additional variable, *status*, that is used as follows. If a process is clean (*i.e.*, not involved into any freezing operation), then its *status* is C. Otherwise, it has status EB or EF and no neighbor can select it as its parent. These two latter states are actually used to perform a *Propagation of Information with Feedback (PIF)* [Cha82, Seg83] in the abnormal trees. Therefore, status EB means "Error Broadcast" and EF means "Error Feedback". From an abnormal root, the status EB is broadcast down in the tree. Then, once the EB-wave reaches a leaf, the leaf initiates a convergecast EF-wave. Once the EF-wave reaches the abnormal root, the tree is said to be *dead*, meaning that there is no process of status C in the tree and no other process can join it. So, the tree can be safely reset from the abnormal root toward the leaves.

Notice that the new variable *status* may also get arbitrary initialization. Thus, we enforce previously introduced predicates as follows.

A self root must have status C, otherwise it is an abnormal root:

$$SelfRootOk(p) \equiv SelfRootOk'(p) \wedge (p.status = \text{C})$$

To be a real child of q, p should have a status coherent with the one of q. This is ex-

Algorithm 4 – Actions of Process p in Algorithm $\mathcal{LE}$.

Inputs.

- $p.id \in \mathfrak{id}$
- $p.\mathcal{N}$

Variables.

- $p.idRoot \in \mathfrak{id}$
- $p.level \in \mathbb{N}$
- $p.par \in p.\mathcal{N} \cup \{p\}$
- $p.status \in \{\text{C}, \text{EB}, \text{EF}\}$

Functions.

$$
\begin{array}{lcl}
Children(p) & \equiv & \{q \in p.\mathcal{N} : q.par = p\} \\
RealChildren(p) & \equiv & \{q \in Children(p) : KinshipOk(q,p)\} \\
Min(p) & \equiv & \min_{\preceq} \{q \in p.\mathcal{N} : q.status = \text{C}\}
\end{array}
$$

Predicates.

$$
\begin{array}{lcl}
p \preceq q & \equiv & (p.idRoot \leq q.idRoot) \wedge [(p.idRoot = q.idRoot) \Rightarrow (p.id \leq q.id)] \\
SelfRoot(p) & \equiv & p.par = p \\
SelfRootOk(p) & \equiv & (p.level = 0) \wedge (p.idRoot = p.id) \wedge (p.status = \text{C}) \\
GoodIdRoot(s,f) & \equiv & (s.idRoot \geq f.idRoot) \wedge (s.idRoot < s.id) \\
GoodLevel(s,f) & \equiv & (s.idRoot = f.idRoot) \Rightarrow (s.level = f.level + 1) \\
GoodStatus(s,f) & \equiv & [(s.status = \text{EB}) \Rightarrow (f.status = \text{EB})] \\
 & & \quad \wedge\ [(s.status = \text{EF}) \Rightarrow (f.status \neq \text{C})] \\
 & & \quad \wedge\ [(s.status = \text{C}) \Rightarrow (f.status \neq \text{EF})] \\
KinshipOk(s,f) & \equiv & GoodIdRoot(s,f) \wedge GoodLevel(s,f) \wedge GoodStatus(s,f) \\
AbnormRoot(p) & \equiv & [SelfRoot(p) \wedge \neg SelfRootOk(p)] \\
 & & \quad \vee\ [\neg SelfRoot(p) \wedge \neg KinshipOk(p)(p.par)] \\
Allowed(p) & \equiv & \forall q \in Children(p), (\neg KinshipOk(q,p) \Rightarrow q.status \neq \text{C})
\end{array}
$$

Guards.

$$
\begin{array}{lcl}
EBroadcast(p) & \equiv & (p.status = \text{C}) \wedge [AbnormRoot(p) \vee (p.par.status = \text{EB})] \\
EFeedback(p) & \equiv & (p.status = \text{EB}) \wedge (\forall q \in RealChildren(p), q.status = \text{EF}) \\
Reset(p) & \equiv & (p.status = \text{EF}) \wedge AbnormRoot(p) \wedge Allowed(p) \\
Join(p) & \equiv & (p.status = \text{C}) \wedge [\exists q \in p.\mathcal{N}, (q.idRoot < p.idRoot) \\
 & & \quad \wedge\ (q.status = \text{C})] \wedge Allowed(p)
\end{array}
$$

Actions.

$$
\begin{array}{llll}
\textbf{EB} :: & EBroadcast(p) & \rightarrow & p.status := \text{EB} \\
\textbf{EF} :: & EFeedback(p) & \rightarrow & p.status := \text{EF} \\
\textbf{R} :: & Reset(p) & \rightarrow & p.status := \text{C} \\
 & & & p.par := p \\
 & & & p.idRoot := p.id \\
 & & & p.level := 0 \\
\textbf{J} :: & Join(p) \wedge \neg EBroadcast(p) & \rightarrow & p.par := Min(p) \\
 & & & p.idRoot := p.par.idRoot \\
 & & & p.level := p.par.level + 1
\end{array}
$$

pressed with the predicate $GoodStatus(p, q)$ which is used to enforce the $KinshipOk(p, q)$ relation:

$$\begin{aligned} GoodStatus(p,q) \quad &\equiv \quad [(p.status = \text{EB}) \Rightarrow (q.status = \text{EB})] \\ &\qquad \wedge [(p.status = \text{EF}) \Rightarrow (q.status \neq \text{C})] \\ &\qquad \wedge [(p.status = \text{C}) \Rightarrow (q.status \neq \text{EF})] \\ KinshipOk(p,q) \quad &\equiv \quad KinshipOk'(p,q) \wedge GoodStatus(p,q) \end{aligned}$$

Precisely, when p has status C, its parent must have status C or EB (if the EB-wave is not propagated yet to p). If p has status EB, its parent must be of status EB because p gets status EB from its parent and its parent will change its status to EF only after p gets status EF. Finally, if p has status EF, its parent can have status EB (if the EF-wave is not propagated yet to its parent) or EF.

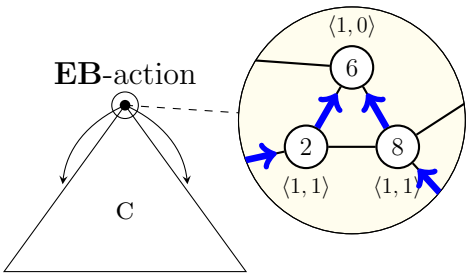
Normal Execution. Remark that, after all abnormal trees have been removed, all processes have status C and the algorithm works as in the initial non self-stabilizing version. Notice that the guard of **J**-action has been enforced so that only processes with status C and which are not abnormal root can execute it, and when executing **J**-action, a process can only choose a neighbor of status C as parent. Moreover, remark that the cleaning of all abnormal trees does not ensure that all fake IDs have been removed. Rather, it guarantees the removal of all fake IDs smaller than ℓ. This implies that (at least) ℓ is a self root at the end of the cleaning and all other processes will elect ℓ within the next $\mathcal{D}$ rounds.

Cleaning Abnormal Trees. We detail now the cleaning of abnormal trees. Figure 4.5 illustrates this cleaning. In the first phase (see Figure 4.5a), the root broadcasts status EB down to its (abnormal) tree: all the processes in this tree execute **EB**-action, switch to status EB and are consequently informed that they are in an abnormal tree.

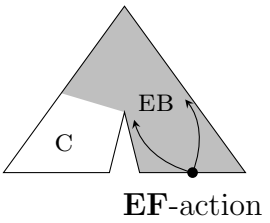
The second phase starts when the EB-wave reaches a leaf. Then, a convergecast wave of status EF is initiated thanks to action **EF**-action (see Figure 4.5b). The system is asynchronous, hence all the processes along some branch can have status EF before the broadcast of the EB-wave is done into another branch. In this case, the parent of these two branches waits that all its children in the tree (processes in the set $RealChildren$) get status EF before executing **EF**-action (Figure 4.5c). When the root gets status EF, all processes in the tree have status EF: the tree is dead.

Then (third phase), the root can reset (safely) to become a self root by executing **R**-action (Figure 4.5e). Its former real children (of status EF) become themselves abnormal roots of dead trees (Figure 4.5f) and reset, *etc.*

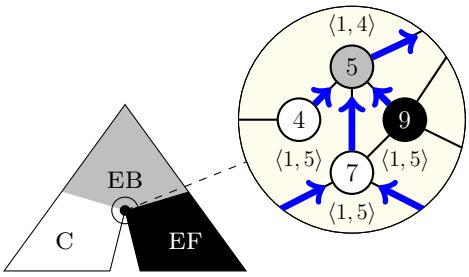
Finally, we used the predicate $Allowed(p)$ to temporarily lock the parent of p in two particular situations — illustrated in Figure 4.6 — where p is enabled to switch its status from C to EB. These locks impact neither the correctness nor the complexity of $\mathcal{LE}$. Rather, they allow us to simplify the proofs by ensuring that, once enabled, **EB**-action remains continuously enabled until executed.



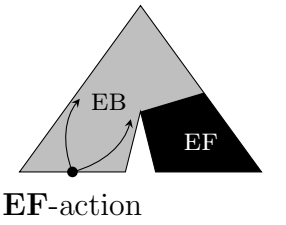
(a) When an abnormal root detects an error, it executes **EB**-action. The EB-wave is broadcast to the leaves. Here, 6 is an abnormal root because it is a self root and its *idRoot* is different from its ID ($1 \neq 6$).



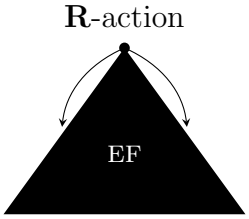
(b) When the EB-wave reaches a leaf, it executes **EF**-action. The EF-wave is propagated up to the root.



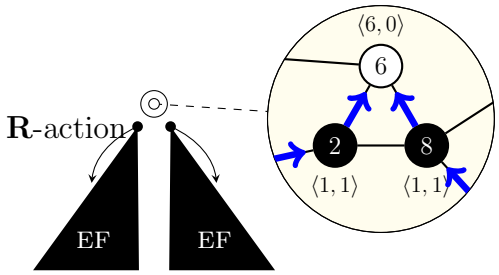
(c) It may happen that the EF-wave reaches a node, here process 5, even though the EB-wave is still broadcasting into some of its proper subtrees: 5 must wait that the status of 4 and 7 become EF before executing **EF**-action.



(d) EB-wave has been propagated in the other branch. An EF-wave is initiated by the leaves.

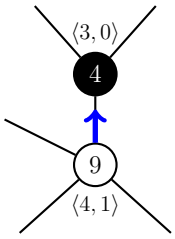


(e) EF-wave reaches the root. The root can safely reset (**R**-action) because its tree is dead. The cleaning wave is propagated down to the leaves.

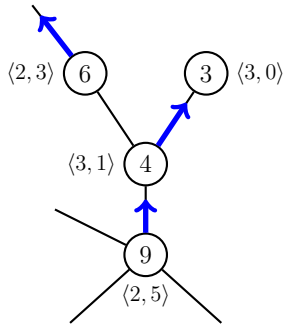


(f) Its children become themselves abnormal roots of dead trees and can execute **R**-action: 2 and 8 can clean because their status is EF and their parent has status C.

Figure 4.5 – Schematic example of the cleaning mechanism of an abnormal tree. Trees and nodes are filled according to the status of their processes: white for C, gray for EB, black for EF.



(a) 4 and 9 are abnormal roots. If 4 executes **R**-action before 9 executes **EB**-action, the kinship relation between 4 and 9 becomes correct and 9 is no more an abnormal root. Then, **EB**-action is no more enabled at 9.



(b) 9 is an abnormal root and $Min(4)$ is 6. If 4 executes **J**-action before 9 executes **EB**-action, the kinship relation between 4 and 9 becomes correct and 9 is no more an abnormal root. Then, **EB**-action is no more enabled at 9.

Figure 4.6 – Example of situations where the parent of a process is locked.

4.3.2 Correctness and Step Complexity

In this section, we prove the correctness and the step complexity of $\mathcal{LE}$ (Theorem 4.3). We first define some useful notions for the proofs. Then, we show that $\mathcal{LE}$ converges to a terminal configuration in finite time (Theorem 4.1) by counting how many times each action is executed. Finally, we prove that a terminal configuration satisfies the specification of the leader election problem (Theorem 4.2).

Some Definitions. First, we instantiate the function $Leader(p)$ used in the specification of the leader election (Section 4.2.2).

Definition 4.2 (*Leader*)

For each process p, for every configuration γ, the value $Leader(p)$ in γ is $\gamma(p).idRoot$.

The rest of the paragraph is dedicated to introducing and justifying the notion of trees induced by the $KinshipOk$ relation. We first show that the predicate $KinshipOk$ is an acyclic relation. To that goal, we define the graph induced by the $KinshipOk$ relation.

Definition 4.3 (*Graph of Kinship Relations*)

For some configuration γ, let $\mathcal{G}_{kr} = (V, KR)$ be a directed graph such that $(p, q) \in KR \Leftrightarrow (\{p, q\} \in E) \wedge (p.par = q) \wedge KinshipOk(p, q)$. $\mathcal{G}_{kr}$ is called the *graph of kinship relations* in γ.

We first show that $\mathcal{G}_{kr}$ is a DAG (Directed Acyclic Graph).

Lemma 4.1

Let γ be a configuration. The graph of kinship relations in γ contains no cycle.

Proof: By definition, for all pairs of processes (p, q) such that $KinshipOk(p, q)$ holds, we have: $p.idRoot \geq q.idRoot$ and $p.idRoot = q.idRoot \Rightarrow p.level = q.level + 1$. Hence, the

processes along any path in $\mathcal{G}_{kr}$ are ordered w.r.t. the strict lexical order on the pair $(idRoot, level)$. The result directly follows. ■

Hence $\mathcal{G}_{kr}$ is a DAG (Directed Acyclic Graph) and even a spanning forest since the condition $p.par = q$ implies at most one successor per process in KR. Below, we define the roots and trees of this spanning forest.

Definition 4.4 (*Root*)

For some configuration γ, a process p satisfies $Root(p)$ (and is called a *root* in γ) if and only if $SelfRoot(p) \vee AbnormRoot(p)$, or equivalently if $SelfRoot(p) \vee \neg KinshipOk(p, p.par)$ holds in γ.

Next, we define the paths, called *KPaths*, that follow the tree structures in $\mathcal{G}_{kr}$, *i.e.*, the paths linking each process to the root of its own tree.

Definition 4.5 (*KPath*)

For every process p, $KPath(p)$ is the unique path $p_0, p_1, \ldots, p_k$ such that $p_k = p$ and satisfying the following conditions:

- $\forall i, 1 \leq i \leq k, (p_i.par = p_{i-1}) \wedge KinshipOk(p_i, p_{i-1})$
- $Root(p_0)$

Using Definitions 4.4 and 4.5, we formally define trees as follows.

Definition 4.6 (*Tree*)

For some configuration γ, for every process p such that $Root(p)$, we define $Tree(p)$, the tree rooted at p, as follows:

$$Tree(p) = \{q \in V : p \text{ is the initial extremity of } KPath(q)\}$$

This means, in particular, that we identify each tree with the ID of its root.

We give in Observation 4.1 an invariant on KPaths when looking at the status of the processes. This property is based on the notion of S-Trace defined below.

Definition 4.7 (*S-Trace*)

For some configuration γ, for a sequence of processes $p_0, p_1, \ldots, p_k$, we define:

$$S\text{-}Trace(p_0, p_1, \ldots, p_k) \in \{\text{C}, \text{EB}, \text{EF}\}^*$$

as the sequence $(\gamma(p_0).status).(\gamma(p_1).status) \ldots (\gamma(p_k).status)$.

Observation 4.1

For any configuration, we have:

$$\forall p \in V, S\text{-}Trace(KPath(p)) \in \text{EB}^*\text{C}^* \cup \text{EB}^*\text{EF}^*.$$

Proof: Let p be a process. If $|KPath(p)| = 1$, Observation 4.1 trivially holds. For $|KPath(p)| \geq 2$, assume by contradiction that $S\text{-}Trace(KPath(p)) \notin \text{EB}^*\text{C}^* \cup \text{EB}^*\text{EF}^*$. Then, $\exists s, f \in KPath(p)$ such that $s.par = f$ and $S\text{-}Trace(f, s) \in \{\text{C.EB}, \text{C.EF}, \text{EF.EB}, \text{EF.C}\}$. In all cases, $\neg GoodStatus(s, f)$ holds and so $\neg KinshipOk(s, f)$ also holds. This contradicts Definition 4.5. ■

Abnormal Trees. Then, we introduce some notions that refine the concept of trees and we prove some preliminary results on the behavior of abnormal trees.

Definition 4.8 (*Normal/Abnormal Tree*)

For every configuration γ and every process p, any tree rooted at p such that $\neg AbnormRoot(p)$ holds in γ is called a *normal tree*. In this case, $SelfRoot(p) \wedge SelfRootOk(p)$ holds in γ, by Definition 4.4.

Any tree that is not normal is said to be *abnormal*.

Definition 4.9 (*Alive/Dead*)

Let γ be a configuration. A process p is called *alive* in γ if and only if $\gamma(p).status = \text{C}$. Otherwise, p is said to be *dead*. A tree $\mathcal{T}$ in γ is called an *alive tree* in γ if and only if $\exists p \in \mathcal{T}$ such that p is alive in γ. Otherwise, it is called a *dead tree*.

Definition 4.10 (*Leave/Join a Tree*)

Let $\gamma \mapsto \gamma'$ be a step. If a process p is in a tree $\mathcal{T}$ in γ, but in a different tree $\mathcal{T}'$ in γ' (namely, the roots of $\mathcal{T}$ and $\mathcal{T}'$ are different), we say that p *leaves* $\mathcal{T}$ and *joins* $\mathcal{T}'$ in $\gamma \mapsto \gamma'$.

Remark 4.1

No process can join a dead tree.

Lemma 4.2

No alive abnormal root can be created.

Proof: Let p be a process which is not an alive abnormal root in some configuration γ. This means that p is dead, p is a normal root ($SelfRoot(p) \wedge SelfRootOk(p)$ holds in γ), or p is not a root ($KinshipOk(p, p.par)$ holds in γ).

Let $\gamma \mapsto \gamma'$ be a step. If p executes **EB**-action in $\gamma \mapsto \gamma'$ (respectively **EF**-action), then $\gamma'(p).status = \text{EB}$ (respectively $\gamma'(p).status = \text{EF}$) and, consequently, p is dead in γ'.

If p executes **R**-action, the predicate $SelfRoot(p) \wedge SelfRootOk(p)$ holds in γ'. So, p is a normal root in γ'.

If p executes **J**-action, let $q = Min(p)$ in γ. By definition of **J**-action, $\gamma(p).idRoot \leq p.id$ (since p is not an abnormal root at γ), $\gamma(q).status = \text{C}$, and $\gamma(p).status = \gamma'(p).status = \text{C}$. Also, $\neg SelfRoot(p)$ holds in γ'.

- If q does not move in $\gamma \mapsto \gamma'$, then $\gamma'(p).par = q$, $\gamma'(q).status = \gamma'(p).status = \text{C}$, $\gamma'(p).level = \gamma(q).level + 1 = \gamma'(q).level + 1$, and $\gamma'(p).idRoot = \gamma(q).idRoot =$

$\gamma'(q).idRoot < \gamma(p).idRoot \leq p.id$. Hence, the predicate $KinshipOk(p, p.par)$ is TRUE in γ'. Now, we already know that $\neg SelfRoot(p)$ holds in γ'. Thus, $\neg SelfRoot(p) \wedge KinshipOk(p, q)$ holds in γ': p is not a root in γ', by Definition 4.4.

- Assume now that q moves during the step $\gamma \mapsto \gamma'$. As $\gamma(q).status = \text{C}$, q can only execute **EB**-action or **J**-action in the step. Consequently, $\gamma'(q).idRoot \leq \gamma(q).idRoot$. Then, $\gamma'(p).idRoot = \gamma(q).idRoot \geq \gamma'(q).idRoot$ and $\gamma'(p).idRoot = \gamma(q).idRoot < \gamma(p).idRoot \leq p.id$. So, the predicate $GoodIdRoot(p, q)$ holds in γ'.

  If q executes **J**-action, then $\gamma'(p).idRoot \neq \gamma'(q).idRoot$. Otherwise, q executes **EB**-action, so $\gamma'(p).idRoot = \gamma'(q).idRoot$ and $\gamma'(p).level = \gamma(q).level + 1 = \gamma'(q).level + 1$. Hence, $GoodLevel(p, q)$ holds in γ'.

  Finally, $\gamma'(q).status \in \{\text{C}, \text{EB}\}$ and $\gamma'(p).status = \gamma(p).status = \text{C}$, so the predicate $GoodStatus(p, q)$ holds in γ'.

  Thus, $\neg SelfRoot(p) \wedge KinshipOk(p, q)$ holds in γ' and, so, p is not a root in γ', by Definition 4.4.

Assume now that p executes no action in the step $\gamma \mapsto \gamma'$. The only way for p to become an alive abnormal root is that $\gamma(p).par$ moves during the step, since the property "alive abnormal root" only depends on p and $p.par$. Furthermore, as p is not an alive abnormal root, when p is a normal root in γ, it stays so, in γ'.

Therefore, let us consider the case when p is not a root in γ and $\gamma(p).par$ moves. As p changes none of its variables, the only way for it to become an alive abnormal root is to have status C in γ and thus in γ'. As $GoodStatus(p, p.par)$ holds in γ, this implies that the status of $\gamma(p).par$ is either EB or C. Looking at case EB, p is a real child of $p.par$ in γ with status C; hence **EF**-action is disabled for $p.par$ in γ. Looking at case C, $p.par$ can execute **EB**-action and can only change its status to EB in $\gamma \mapsto \gamma'$: $GoodStatus(p, p.par)$ holds in γ' and consequently $KinshipOk(p, p.par)$ holds in γ'. $p.par$ can also execute **J**-action in $\gamma \mapsto \gamma'$. This means that in γ and in γ', $p.par$ has status C, hence $GoodStatus(p, p.par)$ holds in γ'. Furthermore, $p.par$ has a smaller value of $idRoot$ in γ', so $GoodIdRoot(p, p.par)$ and $GoodLevel(p, p.par)$ are satisfied in γ', and consequently $KinshipOk(p, p.par)$ holds in γ'. ■

Lemma 4.3

No alive abnormal tree can be created.

Proof: Let $\gamma \mapsto \gamma'$ be a step. Let $p \in V$. Assume there is no alive abnormal tree rooted at p in γ. In particular, p is not an alive abnormal root in γ. Then, assume, by contradiction, that $Tree(p)$ exists and is an alive abnormal tree in γ'.

If $\gamma'(p).status = \text{EF}$, then every process in the tree has status EF (Observation 4.1) and the tree is dead, a contradiction.

If $\gamma'(p).status = \text{C}$, then p is an alive abnormal root in γ'. But no alive abnormal root is created (Lemma 4.2), a contradiction.

If $\gamma'(p).status = \text{EB}$. Then, according to the algorithm, there are two possible cases:

- $\gamma(p).status = \text{EB}$:
  - If $AbnormRoot(p)$ holds in γ, then $Tree(p)$ is dead in γ (otherwise, $Tree(p)$ is an abnormal alive tree in γ, a contradiction). By the definition of **J**-action, no process can join $Tree(p)$ in $\gamma \mapsto \gamma'$.

Moreover, as $\gamma(p).status = \text{EB}$, no process q in $Tree(p)$ satisfies $Reset(q)$ in γ, by Observation 4.1. Consequently, no process can leave $Tree(p)$ in $\gamma \mapsto \gamma'$. So, every process in $Tree(p)$ still have status EF or EB in γ', *i.e.*, $Tree(p)$ is still dead in γ', a contradiction.

– If $\neg AbnormRoot(p)$ holds in γ, then p does not satisfy $SelfRoot(p)$. Indeed, the predicate $SelfRootOk(p)$ implies that $p.status = \text{C}$ in γ, a contradiction. So, let $q = \gamma(p).par \in p.\mathcal{N}$. $\neg AbnormRoot(p)$ in γ implies that $q.status = \text{EB}$ and the predicate $KinshipOk(p, q)$ holds in γ. This latter also implies that $p \in RealChildren(q)$ in γ. Now, $p \in RealChildren(q)$ and $p.status = \text{EB}$ in γ implies that $\gamma(q).status = \text{EB}$ and so q is disabled in γ. Moreover, as $\gamma'(p).status = \text{EB}$, p does not execute any action in $\gamma \mapsto \gamma'$. So, the predicate $\neg AbnormRoot(p)$ still holds in γ', a contradiction.

- $\gamma(p).status = \text{C}$: $\neg AbnormRoot(p)$ holds in γ (otherwise p is an abnormal alive root in γ). Then, p executes **EB**-action in $\gamma \mapsto \gamma'$ to get status EB. So, $EBroadcast(p) \wedge \neg AbnormRoot(p)$ implies that $p.par \neq p$ and $p.par.status = \text{EB}$ in γ. Let $q = \gamma(p).par$. Now, $p.par \neq p$ and $\neg AbnormRoot(p)$ implies that $KinshipOk(p, q)$ holds in γ. So, $p \in RealChildren(q)$ and, as $\gamma(p).status = \text{C}$ and $\gamma(q).status = \text{EB}$, q is disabled in γ. Moreover, as $\gamma'(p).status = \text{EB}$, p necessarily executes **EB**-action in $\gamma \mapsto \gamma'$ which only changes its status to EB. So, $\neg AbnormRoot(p)$ still holds in γ', a contradiction. ■

Finite number of J-actions. To show that every process p executes only a finite number of **J**-actions, we prove below that p can only execute a finite number of **J**-actions in each segment of execution — a segment being separated from its follower by the death or the disappearance of some abnormal alive tree.

Definition 4.11 (*Disappear/Die*)

Let $\gamma \mapsto \gamma'$ be a step and let p be a process such that $Root(p)$ in γ.

- $Tree(p)$ *disappears* during the step $\gamma \mapsto \gamma'$ if and only if $Tree(p)$ is no more defined in γ' — namely $Root(p)$ does not hold in γ'.
- $Tree(p)$ *dies* during the step $\gamma \mapsto \gamma'$ if and only if $Tree(p)$ is alive in γ, yet $Tree(p)$ exists — namely $Root(p)$ holds — and is dead in γ'.

Definition 4.12 (*Segment of Execution*)

Let $e = \gamma_0\gamma_1 \ldots$ be any execution. $e' = \gamma_i \ldots \gamma_j$ is a *(segment) of execution* e if and only if e' is a maximal factor of e, where no abnormal alive tree dies nor disappears.

Figure 4.7 illustrates Definition 4.12. We now show that the number of segments is finite.

Lemma 4.4

There are at most $n + 1$ segments in any execution.

Proof: In the initial configuration, there are at most n abnormal roots (every process) and, consequently, at most n abnormal trees. As no alive abnormal tree can be created

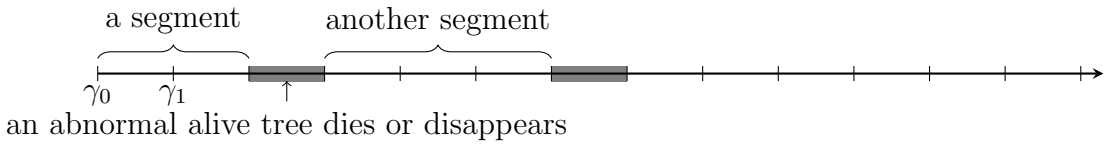


Figure 4.7 – Segments of execution.

(Lemma 4.3), if an abnormal tree is alive, then it is alive since the initial configuration. So, there is at most n trees that die or disappear and, consequently, there are at most $n+1$ segments in the execution. ■

From Lemma 4.4, we have the following remark:

Remark 4.2

There are at most n steps outside segments (more precisely, the steps where at least one abnormal tree dies or disappears) and these steps necessarily contains an execution of **EB***-action.*

We now count the number of **J**-actions processes can execute in a given segment. For that purpose, we first need to prove intermediate lemmas that identify properties on computation steps.

Observation 4.2

Let γ be a configuration and let p a process such that $Reset(p)$ is TRUE *in γ. Then, $Tree(p)$ exists and is dead in γ.*

Proof: Let γ be a configuration and let p be a process such that $Reset(p)$ is TRUE in γ. By definition, $AbnormRoot(p)$ holds in γ, hence $Tree(p)$ is defined in γ. Furthermore, $\gamma(p).status =$ EF: by Observation 4.1, every process in $Tree(p)$ has status EF in γ, and we are done. ■

Lemma 4.5

Let $\gamma \mapsto \gamma'$ be a step and let p be a process such that $p.status \in \{\text{EB}, \text{EF}\}$ in γ. Let $\mathcal{T}$ be the tree which contains p in γ.

1. *$\mathcal{T}$ is an abnormal tree in γ.*
2. *If $\mathcal{T}$ does not disappear during the step $\gamma \mapsto \gamma'$, p is still in $\mathcal{T}$ in γ' unless $\mathcal{T}$ was dead in γ.*

Proof: Let $\gamma \mapsto \gamma'$ be a step and let p be a process such that $p.status \in \{\text{EB}, \text{EF}\}$ in γ. We note r the root of the tree containing p in γ. As $S\text{-}Trace(KPath(p)) \in \text{EB}^*\text{EF}^*$, by Observation 4.1, the status of r in γ is either EF or EB. Hence $AbnormRoot(r)$ holds in γ: $Tree(r)$ is an abnormal tree in γ.

Assume now that $Root(r)$ holds in γ' (the tree does not disappear during the step). If r executes **R**-action in $\gamma \mapsto \gamma'$, Observation 4.2 applies in γ and proves that $Tree(r)$ is dead in γ.

If r does not (or cannot) execute **R**-action, its only possible action is **EF**-action. As $Root(r)$ holds in γ', r is still abnormal root in γ'. Let then $q \in KPath(p)$ in γ with $q \neq r$. By Observation 4.1, $\gamma(q).status \in \{\text{EB}, \text{EF}\}$ also. If $\gamma(q).status = \text{EB}$, q can only execute **EF**-action and if $\gamma(q).status = \text{EF}$, q is disabled, as $q \neq r$. Executing **EF**-action preserves *GoodStatus* and hence preserves also *KinshipOk* relations. Therefore, the $KPath$ from p to r is the same in γ and γ' and then $p \in Tree(r)$ in γ'. ■

Lemma 4.6

Let p be a process and let $\gamma \mapsto \gamma'$ be a step. If p is an abnormal root of status C *in γ, then it is still an abnormal root in γ'.*

Proof: Let $\gamma \mapsto \gamma'$ be a step and let p be a process such that $AbnormRoot(p) \wedge p.status = \text{C}$ in γ: p can only execute **EB**-action. Therefore, $\gamma'(p).status \in \{\text{C}, \text{EB}\}$ and every other variable of p has identical value in γ and γ'.

So, if $SelfRoot(p)$ holds in γ, then $SelfRootOk(p)$ is FALSE in γ, and $SelfRoot(p) \wedge \neg SelfRootOk(p)$ still holds in γ'.

Otherwise, $\neg SelfRoot(p)$ holds in γ, *i.e.*, $p.par \neq p$. Then, $\neg SelfRoot(p)$ still holds in γ'. Let $q \in V$ such that $q = \gamma(p).par = \gamma'(p).par$ and consider the following cases:

- $\gamma(q).status = \text{EF}$: Then, $\neg GoodStatus(p, q)$ holds in γ which implies that $\neg KinshipOk(p, q)$ holds in γ. However, $p \in Children(q)$ in γ. So, $\neg Allowed(q)$ holds in γ, and q is disabled. So, $\gamma'(p).status \in \{\text{C}, \text{EB}\}$ and $\gamma'(q).status = \text{EF}$, which implies that the predicate $\neg GoodStatus(p, q)$ holds in γ'. Thus, we have $\neg KinshipOk(p, q)$ in γ'.

- $\gamma(q).status = \text{EB}$: Then, $GoodStatus(p, q)$ holds in γ. So, $AbnormRoot(p)$ in γ implies that the predicate $\neg GoodIdRoot(p, q) \vee \neg GoodLevel(p, q)$ holds in γ. Now, q can only executes **EF**-action in $\gamma \mapsto \gamma'$. So, neither p nor q modify their variables *par*, *idRoot*, or *level* in $\gamma \mapsto \gamma'$, and, consequently, $\neg GoodIdRoot(p, q) \vee \neg GoodLevel(p, q)$ still holds in γ'. So, $\neg KinshipOk(p, q)$ holds in γ'.

- $\gamma(q).status = \text{C}$: As $AbnormRoot(p)$ holds in γ, $\neg KinshipOk(p, q)$ in γ. Thus, $\neg Allowed(q)$ holds in γ because $p \in Children(q)$ and $p.status = \text{C}$ in γ. So, q cannot execute **J**-action in $\gamma \mapsto \gamma'$. Then, $\gamma(q).status = \text{C}$ and $\gamma(p).status = \text{C}$ implies that $GoodStatus(p, q)$ holds in γ. So, $AbnormRoot(p)$ in γ implies that $\neg GoodIdRoot(p, q) \vee \neg GoodLevel(p, q)$ holds in γ. As p and q can only modify their status during the step $\gamma \mapsto \gamma'$ (q can only execute **EB**-action in $\gamma \mapsto \gamma'$), $\neg GoodIdRoot(p, q) \vee \neg GoodLevel(p, q)$ still holds in γ'. So, $\neg KinshipOk(p, q)$ holds in γ'.

In any cases, $\neg KinshipOk(p, q)$ holds in γ'. As the predicate $\neg SelfRoot(p)$ holds in γ', $AbnormRoot(p)$ holds in γ'. ■

Lemma 4.7

*Let γ be a configuration and let p be a process such that $\gamma(p).status \in \{\text{EB}, \text{EF}\}$. Let $\mathcal{T}$ be the tree which contains p in γ. Let γ_R be the first configuration, if any, after γ, such that p executes an **R**-action $\gamma_R \mapsto \gamma_{R+1}$.*

Assume γ_R exists, then $\mathcal{T}$ is dead in γ_R or has disappeared (at least once) between γ and γ_R.

Proof: Let γ be a configuration and let p be a process such that $p.status \in \{\text{EB}, \text{EF}\}$ in γ. We note r the root of the tree which contains p in γ. Let $\gamma = \gamma_0\gamma_1...$ be an execution starting at γ. Let γ_R be the first configuration, if any, in this execution such that p executes an **R**-action during the step $\gamma_R \mapsto \gamma_{R+1}$. Assume γ_R exists.

For every configuration γ_x, $x \in \{0, ..., R-1\}$, the status of p is EB or EF. Hence, Lemma 4.5 applies iteratively in γ_x: either $Tree(r)$ disappears during the step $\gamma_x \mapsto \gamma_{x+1}$, or, if not, $p \in Tree(r)$ in γ_{x+1}. Hence, in γ_R, either $Tree(r)$ has disappeared or, if not, $p \in Tree(r)$.

When $p \in Tree(r)$ in γ_R, by assumption, p executes an **R**-action between γ_R and γ_{R+1}. Hence, $AbnormRoot(p)$ holds in γ_R and thus $p = r$. Furthermore, Observation 4.2 applies and proves that $Tree(r)$ is dead in γ_R. ■

Lemma 4.8

Let p be a process and let $\gamma \mapsto \gamma'$ be a step. Let $\mathcal{T}$ be the tree that contains p in γ. If $EBroadcast(p)$ holds in γ, then $\mathcal{T}$ is an abnormal alive tree in γ and, if $\mathcal{T}$ has not disappeared in γ', p still belongs to $\mathcal{T}$ in γ'.

Proof: Let $\gamma \mapsto \gamma'$ be a step. Let p be a process such that $EBroadcast(p)$ holds in γ. We note r the root of the tree which contains p in γ.

If $AbnormRoot(p)$ holds in γ, then $p = r$ is the root of an alive abnormal tree, since $\gamma(p).status = \text{C}$. Furthermore, if $Tree(p)$ exists in γ', $p \in Tree(p)$ in γ', trivially.

Otherwise, $\neg AbnormRoot(p)$, $p.par.status = \text{EB}$, and $KinshipOk(p, p.par)$ holds in γ. Applying Lemma 4.5 to $\gamma(p).par$, we have that $\gamma(p).par$ belongs to an abnormal alive tree in γ and so does p: $Tree(r)$ is an alive abnormal tree.

Furthermore, first note that $\gamma(p).par = \gamma'(p).par$ (p can only change its status to EB in $\gamma \mapsto \gamma'$: either p do not move or executes **EB**-action). So, still by Lemma 4.5, in γ', if $Tree(r)$ exists in γ', $\gamma'(p).par$ belongs to $Tree(r)$ in γ', since $Tree(r)$ is not dead in γ ($\gamma(p).status = \text{C}$).

As $KinshipOk(p, p.par)$ holds in γ, we have that $p \in RealChildren(q)$ in γ. Since $\gamma(p).status = \text{C}$, q is disabled in γ (because of p) and, as p can only modify its status to EB in $\gamma \mapsto \gamma'$, we still have $p \in RealChildren(q)$ in γ', *i.e.*, p and q belong to the same abnormal tree, $Tree(r)$, in γ'. ■

Corollary 4.1

Let γ be a configuration and let p be a process such that $EBroadcast(p)$ holds in γ. Let $\mathcal{T}$ the tree which contains p in γ. Let γ_R be the first configuration, if any, since γ, such that p executes an **R***-action $\gamma_R \mapsto \gamma_{R+1}$.*

Assume γ_R exists, then $\mathcal{T}$ is an alive abnormal tree in γ but it is dead in γ_R or has disappeared (at least once) between γ and γ_R.

Proof: Let γ be a configuration and let p be a process such that $EBroadcast(p)$ holds in γ. We note r the root of the tree which contains p in γ. Lemma 4.8 applies in γ: $Tree(r)$ is an alive abnormal tree in γ.

Let $\gamma = \gamma_0\gamma_1...$ be an execution starting at γ. Let γ_R be the first configuration, if any, in this execution such that p executes an **R**-action during the step $\gamma_R \mapsto \gamma_{R+1}$. We assume that γ_R exists. Then at some step, $\gamma_i \mapsto \gamma_{i+1}$, p executes a **EB**-action, with $i < R$.

Lemma 4.8 applies iteratively from γ_0 and to γ_i: either $Tree(r)$ has disappeared in γ_1 (and so between γ_0 and γ_{i+1}), or p stays in $Tree(r)$ in γ_1 (and so between γ_0 and γ_{i+1}), and so on.

If $Tree(r)$ has not yet disappeared in γ_{i+1}, then $p \in Tree(r)$ in γ_{i+1} and $\gamma_{i+1}(p).status = EB$. Here, Lemma 4.7 applies and proves that $Tree(r)$ has disappeared in γ_R or p is in $Tree(r)$ in γ_R. ■

Lemma 4.9

Let p be a process. Let s be a segment of execution. Between any two executions of **J***-action by p in s, p can only execute* **J***-actions.*

Proof: Let $s = \gamma_0\gamma_1\ldots$ be a segment of execution and $p \in V$. Consider two executions of **J**-action by p during s: one in $\gamma_i \mapsto \gamma_{i+1}$ and the other in $\gamma_j \mapsto \gamma_{j+1}$, with $i < j$. Assume by contradiction that p executes an action different from **J**-action between γ_{i+1} and γ_j. Let $\gamma_k \mapsto \gamma_{k+1}$ be the first step between γ_{i+1} and γ_j during which p executes some other action: this is a **EB**-action. Let $\gamma_l \mapsto \gamma_{l+1}$ be the last step between γ_{i+1} and γ_j during which p executes some other action: this is a **R**-action (hence $k < l$).

Now, Lemma 4.1 applies since in γ_k, $EBroadcast(p)$ holds, and in some step later $\gamma_l \mapsto \gamma_{l+1}$, p executes a **R**-action. This proves that in γ_k, some abnormal tree is alive and that in γ_l, this tree is dead or has disappeared. Hence γ_k and γ_l are not in the same segment, a contradiction. ■

Lemma 4.10

In a segment of execution, there are at most $(n-1)(n-2)/2$ executions of **J***-action.*

Proof: Let $p \in V$. First, p only executes **J**-actions between two **J**-actions in the same segment (Lemma 4.9). So, using the guard of **J**-action, it follows that the value of the $p.idRoot$ always decreases during any sequence of **J**-actions which means that p cannot set $p.idRoot$ two times to the same value during the segment.

Let A be the set of processes q such that $q.status = \text{C}$ at the beginning of the segment. Let B the set of processes q such that q executes an **R**-action in the segment. $A \cap B = \emptyset$. Indeed, pick a process $q \in A \cap B$. q switches from status C at the beginning to status EB, and then to status EF since some step later, it executes **R**-action. Hence, there exists a configuration γ_b in the segment such that $EBroadcast(q)$ is TRUE and another γ_r, later on such that **R**-action occurs: hence Corollary 4.1 applies and proves that the tree of q in γ_b is abnormal alive and that it dies or disappears some step before γ_r. This contradicts the definition of segment. Hence, $|A| + |B| \leq n$.

Now, $p.idRoot$ can only be assigned to:

1. values which are present in variables $idRoot$ of processes in A at the first configuration of the segment,
2. ID of processes in B.

Let $f : V \to \mathfrak{id}$ be a function such that $\forall p \in A \cup B$, if $p \in A$, $f(p) = x$, where x is the value of $p.idRoot$ at the beginning of the segment; otherwise, $f(p) = p.id$. Let $p_0, \dots p_{k-1}$ (with $k \leq n$) be the set of processes in $A \cup B$ in ascending order of f. p_i changes at most i times of $idRoot$. Hence, in a given segment, the number of executed **J**-actions, noted $\sharp\mathbf{J}$, satisfies the following inequality:

$$\sharp\mathbf{J} \leq \sum_{i=0}^{k-1} i \leq \sum_{i=0}^{n-1} i = \frac{(n-1)(n-2)}{2}$$

■

By Lemmas 4.4 and 4.10, in any execution, there are at most $n+1$ segments, where processes execute at most $(n-1)(n-2)/2$ **J**-actions. Hence, follows:

Corollary 4.2

*In any execution, there are at most $\frac{n^3}{2} - n^2 + \frac{n}{2} + 1$ **J**-actions executed inside segments.*

Finite Number of Other Actions. Below, we show an upper bound on the number of executions of other actions.

Lemma 4.11

*In any execution, each process can execute at most n **R**-actions.*

Proof: First, by definition, there are at most n abnormal alive trees in the initial configuration. Let $\sharp AbT$ be that number. Moreover, $\sharp AbT$ can only decrease, by Lemma 4.3.

Let p be a process. We first show that when p executes **R**-action for the first time, $\sharp AbT \leq n-1$. Then, we show that after every subsequent execution of a **R**-action by p, $\sharp AbT$ necessarily decreases. Hence, we will conclude that p cannot execute **R**-action more than n, because $\sharp AbT$ cannot be negative.

Consider the first step $\gamma_i \mapsto \gamma_{i+1}$ where p executes **R**-action. Using Observation 4.2, $Tree(p)$ exists and is dead in γ_i. Hence, there are at most $n-1$ abnormal alive trees in γ_i.

Consider the j^{th} execution of **R**-action by p, with $j > 1$. After the $(j-1)^{\text{th}}$ **R**-action of p, the status of p is C. So, between the $(j-1)^{\text{th}}$ and the j^{th} **R**-action, the status

of p thus switches from C to EB and from C to EF, so that p can switch its status from EF to C when executing its j^{th} **R**-action. Hence, meanwhile there exists a configuration γ_b such that $EBroadcast(q)$ is TRUE and another γ_r, later on such that p executes its j^{th} **R**-action in $\gamma_r \mapsto \gamma_{r+1}$: Corollary 4.1 applies and proves that the tree to which p belongs in γ_b is abnormal alive and that tree dies or disappears some step before γ_r, and we are done. ■

Let p be a process. p necessarily executes **R**-action between two executions of **EF**-action (resp. **EB**-action). Hence, we have the following corollary.

Corollary 4.3

In any execution, a process can execute **EB***-action and* **EF***-action at most n times, each.*

By Remark 4.2, Corollaries 4.2, 4.3, and Lemma 4.11:

Theorem 4.1 (*Convergence*)

Every execution contains at most $\frac{n^3}{2} + 2n^2 + \frac{n}{2} + 1$ steps.

Terminal Configurations. We now show that in a terminal configuration, there is one and only one leader process, known by all processes, *i.e.*, for every two processes, p and q, we have $Leader(p) = Leader(q)$ and $Leader(p)$ is the ID of some existing process.

Lemma 4.12

In a terminal configuration, every process has status C.

Proof: By contradiction, consider a terminal configuration γ where some process p satisfies $p.status \neq$ C. Then two cases are possible:

1. $p.status =$ EB. By Observation 4.1, $\exists q \in V$ such that $q.status =$ EB $\wedge$ $(\forall q' \in RealChildren(q), q'.status \neq$ EB$) \wedge p \in KPath(q)$. If $RealChildren(q) = \emptyset$, then q can executes **EF**-action. Otherwise, there are two cases. If $\forall q' \in RealChildren(q)$ then, $q'.status =$ EF and q can execute **EF**-action. Otherwise, there is $q' \in RealChildren(q)$ such that $q'.status =$ C and then q' can execute **EB**-action. Hence, in both cases, γ is not terminal, a contradiction.
2. $p.status =$ EF. By Observation 4.1, $\exists q \in V$ such that $q.status =$ EF $\wedge$ $(Root(q) \vee (KinshipOk(q, q.par) \wedge q.par.status \neq$ EF$) \wedge q \in KPath(p)$.

   If $Root(q)$, then $AbnormRoot(q) \vee SelfRoot(q)$. Now, $q.status =$ EF implies that $AbnormRoot(q)$ holds. So, in all cases, $q.status =$ EF $\wedge$ $AbnormRoot(q)$ holds. If $Allowed(q)$ holds, then **R**-action is enabled at q, a contradiction. Otherwise, $\exists r \in Children(q)$ such that $\neg KinshipOk(r, q) \wedge r.status =$ C. So **EB**-action is enabled at r, a contradiction.

   If $\neg Root(q)$, either $q.par.status =$ C, $AbnormRoot(q)$ holds and we obtain a contradiction as in the case where $Root(q)$ holds, or $q.par.status =$ EB and using the same argument as in case 1, we can deduce that some process is enabled, a contradiction.

   Hence, all cases, γ is not terminal, a contradiction.

Theorem 4.2 (*Correctness*)

In a terminal configuration, $\forall p, q \in V, Leader(p) = Leader(q)$ and $Leader(p)$ is the ID of some existing process.

Proof: Let γ be a terminal configuration. Assume first, by contradiction, that there are at least two leaders. As $\mathcal{G}$ is connected, $\exists p, q \in V$ such that $\gamma(p).leader \neq \gamma(q).leader$ and $q \in p.\mathcal{N}$. Now, assume without loss of generality that, in γ:

$$Leader(p) = \gamma(p).idRoot < \gamma(q).idRoot = Leader(q)$$

By Lemma 4.12, $p.status = q.status =$ C. Then, either $EBroadcast(q)$ is TRUE and q can execute **EB**-action or q can execute **J**-action. Hence γ is not terminal, a contradiction.

Assume now that the leader is not one of the processes, *i.e.*, is a fake ID. Let $p \in V$ such that its *level* is minimum. Notice that $\gamma(p).status =$ C by Lemma 4.12. If $SelfRoot(p)$ holds in γ, $\gamma(p).idRoot \neq p.id$. So, $AbnormRoot(p)$ holds and p can execute **EB**-action. Otherwise, there is $q \in p.\mathcal{N}$ such that $\gamma(p).par = q$. As the level of p is minimum, $\gamma(p).level \leq \gamma(q).level$. So, $AbnormRoot(p)$ holds and p can execute **EB**-action. Hence, γ is not terminal, a contradiction. ■

Using Theorem 4.2, there is exactly one root in a terminal configuration (the leader elected). So the graph of kinship relations in a terminal configuration contains exactly one tree. Hence, we can conclude:

Remark 4.3

In a terminal configuration, $\mathcal{G}_{kr}$ is a spanning tree rooted at the leader.

Theorems 4.1 and 4.2 establish the self-stabilization, silence, and step complexity of Algorithm $\mathcal{LE}$. Moreover, note that *idRoot* can be stored in b bits and *level* can be stored in $\Theta(\log n)$ bits. Hence, we can conclude:

Theorem 4.3

Algorithm $\mathcal{LE}$ is a silent self-stabilizing algorithm w.r.t. *SP_{LE} working under a distributed unfair daemon. Its step complexity is at most $\frac{n^3}{2} + 2n^2 + \frac{n}{2} + 1$ steps. Its memory requirement is $\Theta(\log n + b)$ bits per process.*

4.3.3 Complexity Analysis

In this subsection, we study the complexity of Algorithm $\mathcal{LE}$ in rounds and we make a worst-case analysis of its stabilization time both in steps and rounds.

Stabilization Time in Rounds. First, we study the "good" cases, *i.e.*, when the system is in a clean configuration (defined below). From such configurations, the execution consists in building a tree rooted at ℓ using **J**-action only. Once, the tree is built, the system is in a terminal configuration, where every process has elected ℓ.

Definition 4.13 (*Clean configuration*)

A configuration γ is called a *clean configuration* if and only if for every process p, $\neg EBroadcast(p) \wedge p.status = \text{C}$ holds in γ. A configuration that is not clean is said to be *dirty*.

Remark 4.4

By definition, in a clean configuration, every process p has status C and either p is a normal root, i.e., $SelfRoot(p) \wedge SelfRootOk(p)$, or (exclusively) $KinshipOk(p, p.par)$ holds.

Remark 4.5

Notice that in a clean configuration, the only action a process p can execute is **J***-action, provided that $Join(p)$ holds. Note also that $Allowed(p)$ always holds due to Remark 4.4. Verifying $Join(p)$ then reduces to: $\exists q \in p.\mathcal{N}, (q.idRoot < p.idRoot)$. In this case, the value of $p.idRoot$ can only decrease.*

Lemmas 4.13 to 4.16 proves that, starting from a clean configuration, the system reaches in $O(\mathcal{D})$ rounds a terminal configuration (see Theorem 4.4). We first show the set of clean configurations is closed.

Lemma 4.13

The set of clean configurations is closed.

Proof: Let $\gamma \mapsto \gamma'$ be a step such that γ is a clean configuration. By definition, all processes have status C in γ. So, processes can only execute **J**-action (Remark 4.5) in $\gamma \mapsto \gamma'$, and consequently all processes have status C in γ'. Now, $\forall p \in V, \neg EBroadcast(p) \wedge p.status = \text{C}$ in γ implies that there is no alive abnormal root in γ. By Lemma 4.2, there is no alive abnormal root in γ' too. Now, the fact that all processes have status C and there is no alive abnormal root in γ' implies that $\forall p \in V, \neg EBroadcast(p) \wedge p.status = \text{C}$ in γ', *i.e.*, γ' is clean. ■

Using Lemma 4.13, we show below that if a process is enabled in a clean configuration (for the only action it can execute, *i.e.*, **J**-action) it remains enabled until it executes it.

Lemma 4.14

In a clean configuration, if **J***-action is enabled at p, it remains enabled until it is executed by p.*

Proof: Let $\gamma \mapsto \gamma'$ be a step such that γ is a clean configuration. Assume by contradiction that **J**-action is enabled at p in γ and not in γ', but p did not execute **J**-action between γ and γ'. By Lemma 4.13, γ' is also a clean configuration. So, $\neg EBroadcast(p) \wedge p.status = \text{C}$ holds in γ'.

But join(p) must be FALSE in γ'. Using Remark 4.5, this means that there necessarily exists a neighbor of p, say q, such that $\gamma(q).idRoot < \gamma(p).idRoot$ but $\gamma'(q).idRoot \geq \gamma'(p).idRoot = \gamma(p).idRoot$. This contradicts Remark 4.5. ■

Lemma 4.15

There is no (fake) idRoot smaller than $\ell.id$ in a clean configuration.

Proof: Let γ be a clean configuration. Assume there exists a process of *idRoot* smaller than ℓ. Let p be such a process such that $p.idRoot$ is minimum among all the processes and $p.level$ is minimum among all the processes having *idRoot* minimum.

Note that $p.idRoot \neq p$ so $SelfRootOk(p)$ is FALSE in γ. Hence, using Remark 4.4, the predicate $KinshipOk(p, p.par)$ holds in γ. Since we take p of minimum *idRoot* $p.idRoot \leq p.par.idRoot$ in γ. $GoodIdRoot(p, p.par)$ implies that $p.idRoot \geq p.par.idRoot$, so $p.idRoot = p.par.idRoot$. Now, $GoodLevel(p, p.par)$ implies that $p.level = p.par.level+1$, which contradicts the minimality of $p.level$. ■

For any process p, p can only set $p.idRoot$ to its own ID or copy the value of $idRoot(q)$, where q is one of its neighbors. So, we have the following remark:

Remark 4.6

No fake ID is created during any step.

Lemma 4.16

In a clean configuration, if the idRoot of a process p is $\ell.id$, p is disabled forever.

Proof: Let γ be a clean configuration. Let p be a process with $\gamma(p).idRoot = \ell$. By Remark 4.5, only **J**-action can be enabled in γ. Moreover, its guard reduces to $\exists q \in p.\mathcal{N}, (idRoot(q) < p.idRoot)$. But Lemma 4.15 ensures that this cannot be TRUE, hence p is disabled in γ. Then, by Lemma 4.13 and Remark 4.6, this will be TRUE forever. ■

Corollary 4.4

A clean configuration where $\forall p \in V, p.idRoot = \ell.id$, is terminal.

Theorem 4.4

In a clean configuration, the system reaches a terminal configuration where $\forall p \in V, p.idRoot = \ell$ in at most $\mathcal{D}$ rounds.

Proof: Consider any execution e that starts from a clean configuration. In the following, we denote by ρ_i the first configuration of the i^{th} round in e. We show by induction on the distance $d \geq 0$ between the processes and ℓ that $\forall p \in V$ such that $\|p, \ell\| \leq d$, $\rho_d(p).idRoot = \ell.id$.

Base case: If $\|p, \ell\| = 0$, $p = \ell$. Notice that if the predicate $GoodIdRoot(p, p.par)$ holds in ρ_0, it would implies that $p.idRoot < p.id$ which is FALSE by Lemma 4.15. So $KinshipOk(p, p.par)$ cannot hold in ρ_0. Hence, $SelfRoot(p) \wedge SelfRootOk(p)$ holds in ρ_0 (by Remark 4.4) and $\rho_0(p).idRoot = p.id = \ell.id$.

Induction step: Assume the property holds at some $d \geq 0$. If $\|p, \ell\| = d+1$, $\exists q \in p.\mathcal{N}$ such that $\|q, \ell\| = d$. By induction hypothesis and by Lemma 4.16, $idRoot(q) = \ell.id$ and q is disabled forever since ρ_d.

If $p.idRoot.id = \ell$ in ρ_d, it remains so forever (Lemma 4.16). If $p.idRoot \neq \ell.id$ in ρ_d then $q.idRoot < p.idRoot$ (Lemma 4.15). Then, **J**-action is enabled at p in ρ_d and remains enabled until p executes it (Lemma 4.14). As there is no fake ID smaller than $\ell.id$ (Lemma 4.15), $p.idRoot = \ell.id$ after p executes **J**-action, *i.e.*, after at most one round. Hence, $p.idRoot = \ell.id$ in ρ_{d+1}.

As $\mathcal{D} \geq \max\{\|p, \ell\|, p \in V\}$, in at most $\mathcal{D}$ rounds, the system reaches a configuration where $\forall p \in V, p.idRoot = \ell.id$. By Corollary 4.4, this configuration is terminal. ■

Previously, we proved that, starting from a clean initial configuration, the system reaches a terminal configuration in at most $\mathcal{D}$ rounds. But what happens if the initial configuration is dirty, *i.e.*, if there is a process p such that the predicate $EBroadcast(p)$ holds or $p.status \neq$ C. In this section, we prove that starting from a dirty configuration, the system reaches a clean configuration in at most $3n$ rounds. More precisely, we show that a dirty configuration contains abnormal trees that are "cleaned" in at most $3n$ rounds. The system will be in a clean configuration afterwards.

Lemma 4.17

In a dirty configuration, there exists at least one abnormal root.

Proof: Let γ be a dirty configuration. Then, $\exists p \in V$ such that $p.status \neq \text{C} \vee EBroadcast(p)$. We search for an abnormal root.

1. If $p.status \in \{\text{EB}, \text{EF}\}$, using Observation 4.1, there is $q \in KPath(p)$ such that $q.status \in \{\text{EB}, \text{EF}\} \wedge Root(q)$. Then, $AbnormRoot(q) \vee SelfRoot(q)$ holds in γ. Now, $SelfRoot(q) \wedge q.status \in \{\text{EB}, \text{EF}\}$ implies $AbnormRoot(q)$. Hence, in all cases, $AbnormRoot(q)$ holds.
2. If $EBroadcast(p)$ holds, Lemma 4.8 applies and we are done. ■

We have just shown that there are abnormal roots (and so abnormal trees) in dirty configurations. Below, we prove that these abnormal trees will disappear after three waves of "cleaning". After the first wave, an abnormal tree becomes dead (Theorem 4.5), after the second wave any abnormal root gets the status EF (Theorem 4.6) and finally after the third wave there is no more abnormal trees (Theorem 4.7), hence the system is in a clean configuration.

The following technical lemma is used in the proof of Theorem 4.5.

Lemma 4.18

When **EB***-action is enabled at a process p, it remains enabled until p executes* **EB***-action.*

Proof: Assume that **EB**-action is enabled at a process p in a configuration γ, but p did not execute **EB**-action during the step $\gamma \mapsto \gamma'$. Notice that p does not execute any action during this step, as guards are mutually exclusive. As **EB**-action is enabled in γ, $\gamma(p).status =$ C and then, $\gamma'(p).status =$ C.

First, assume that the predicate $AbnormRoot(p)$ holds in γ. If $SelfRoot(p) \wedge \neg SelfRootOk(p)$ holds in γ and, as these predicates only depends on the local state of p and as p does not execute any action during the step, it also holds in γ': the action is still enabled in γ'. Otherwise, $\neg SelfRoot(p) \wedge \neg KinshipOk(p, p.par)$ holds in γ. These predicates only depends on the local state of p and its parent. Now, $Allowed(p.par)$ does not hold in γ because of p, so $p.par$ cannot execute **R**-action nor **J**-action during $\gamma \mapsto \gamma'$. Then, either $p.par$ executes **EF**-action, changes its status to EF and then, $GoodStatus(p, p.par)$ is FALSE in γ', or $p.par$ executes **EB**-action and changes its status to EB. In these two cases, $EBroadcast(p)$ holds in γ'.

Now, assume $p.par.status =$ EB. $p.par$ can only execute **EF**-action and change its status to EF. Then, the predicate $GoodStatus(p, p.par)$ is FALSE in γ', which implies that $EBroadcast(p)$ holds in γ'. ■

Theorem 4.5

In at most n rounds, the system reaches a configuration where every abnormal tree (if any) is dead.

Proof: Consider any execution $e = \gamma_0, \ldots$. $\forall i > 0$, we denote by γ_{R_i} the last configuration of the i^{th} round and so the first configuration of the $(i+1)^{\text{th}}$ round of e. Moreover, let $\gamma_{R_0} = \gamma_0$ be the initial configuration. We show by induction on the length of the KPaths that, $\forall i \geq R_d$ $(d \geq 1)$, $\forall p \in V$, if p is in an abnormal tree and $|KPath(p)| \leq d$ in γ_i, then p is dead in γ_i.

Base Case: If p is in an abnormal tree and $|KPath(p)| = 1$, p is an abnormal root. As no alive abnormal root is created (Lemma 4.2), if p is alive, it is an alive abnormal root since γ_{R_0} and if predicate $(p.status = C \wedge AbnormRoot(p))$ becomes FALSE in some configuration, then it remains FALSE forever. Hence, it is sufficient to show that any alive abnormal root is no more an alive abnormal root after one round (that is, from γ_{R_1}).

By definition, **EB**-action is enabled at p in γ_{R_0} and p executes **EB**-action during the first round (using Lemma 4.18). Hence, p is dead at the end of the first round, and we are done.

Induction Hypothesis: Let $d \geq 1$. Assume that $\forall i \geq R_d$, $\forall p \in V$, if p belongs to an abnormal tree and $|KPath(p)| \leq d$ in γ_i, then p is dead in γ_i.

Induction Step: We first show that for every $p \in V$, for every $i \geq R_d$, if $(p.status =$ C$\wedge$ $|KPath(p)| \leq d+1)$ is FALSE in configuration γ_i, then for every $j \geq i$, $(p.status =$ C $\wedge\, |KPath(p)| \leq d+1)$ is FALSE in configuration γ_j.

Assume by contradiction that the predicate $(p.status =$ C $\wedge\, |KPath(p)| \leq d+1)$ is FALSE in γ_j, but TRUE in γ_{j+1} $(j \geq i)$. By induction hypothesis, $|KPath(p)| = d+1 > 1$ in γ_{j+1} (indeed, p is alive in γ_{j+1}). So, $\gamma_{j+1}(p).par \neq p$. So, let $q \in p.\mathcal{N}$ such that $\gamma_{j+1}(p).par = q$. By definition, $|KPath(q)| = d$ in γ_{j+1}. By induction hypothesis, $\gamma_{j+1}(q).status \in \{EB, EF\}$. Now, $p.status =$ C and $|KPath(p)| > 1$ in γ_{j+1}, so p is not an abnormal root in γ_{j+1}. Hence, $\gamma_{j+1}(q).status =$ EB (by Observation 4.1) and, consequently, $\gamma_j(q).status \in \{C, EB\}$.

- If $\gamma_j(q).status =$ EB, then p does not execute any action during the step $\gamma_j \mapsto \gamma_{j+1}$ (otherwise, $\gamma_{j+1}(p).status \neq$ C or $\gamma_{j+1}(p).par \neq q$). Hence,

$\gamma_j(p).status = \gamma_{j+1}(p).status = \text{C}$. By hypothesis, '$p.status = \text{C} \wedge |KPath(p)| \leq d+1$ is FALSE in γ_j, so we have $|KPath(p)| > d+1$ in γ_j.
Now, $\gamma_j(p).status = \text{C}$ and $\gamma_j(q).status = \text{EB}$, so $S\text{-}Trace(KPath(p)) = \text{EB}^+\text{C}$ in γ_j (Observation 4.1) and p is the only process in its $KPath$ that can execute an action in $\gamma_j \mapsto \gamma_{j+1}$. Hence, for every q such that $q \in KPath(p)$ in γ_j, $q \in KPath(p)$ in γ_{j+1}, and then $|KPath(p)| > d+1$ in γ_{j+1}. So $p.status = \text{C} \wedge |KPath(p)| \leq d+1$ is FALSE in γ_{j+1}, a contradiction.

- If $\gamma_j(q).status = \text{C}$, then q is in an alive abnormal tree in γ_j (indeed, q executes **EB**-action in $\gamma_j \mapsto \gamma_{j+1}$, and so Lemma 4.8 applies). As q is alive in γ_j, we have $|KPath(q)| > d$ in γ_j by induction hypothesis. Moreover, q is not an abnormal root (there is no more alive abnormal root after the first round, see the base case). Hence, the status of its parent in γ_j is EB.
  Now, $|KPath(q)| > d$ and $S\text{-}Trace(KPath(q)) = \text{EB}^+\text{C}$ in γ_j (Observation 4.1). So, q is the only one in its KPath that executes an action in $\gamma_j \mapsto \gamma_{j+1}$ and this action is **EB**-action, that maintains the $KinshipOk$ relation. Hence, $|KPath(q)| > d$ in γ_{j+1} and consequently, $|KPath(p)| > d+1$ in γ_{j+1}, a contradiction.

Hence, $\forall p \in V$, if $(p.status = \text{C} \wedge |KPath(p)| \leq d+1)$ is FALSE in some configuration γ_i with $i \geq R_d$, then $(p.status = \text{C} \wedge |KPath(p)| \leq d+1)$ remains FALSE forever.

Now, **EB**-action is continuously enabled $\forall p$ such that p is alive $|KPath(p)| = d+1$ in γ_{R_d} (by induction hypothesis and Lemma 4.18). So, p becomes dead during the round and, $\forall j \geq R_{d+1}$, γ_j contains no alive process p such that $|KPath(p)| \leq d+1$.

$n \geq \max\{|KPath(p)| : \forall p \in V\}$. Hence, any process in an abnormal tree becomes dead in at most n rounds. ■

Lemma 4.19

If **EF***-action is enabled at a process p, it remains enabled until p executes* **EF***-action.*

Proof: Let $\gamma \mapsto \gamma'$ be a step. Assume by contradiction **EF**-action is enabled at a process p in γ and is not enabled in γ', but p did not execute **EF**-action during the step $\gamma \mapsto \gamma'$. Notice that p does not execute any action during this step, as guards are mutually exclusive. As $EFeedback(p)$ holds in γ, $\gamma(p).status = \gamma'(p).status = \text{EB}$. As $EFeedback(p)$ does not hold in γ' and no process can execute **J**-action and choose a process of status EB as parent, $\exists q \in RealChildren(p)$ such that $\gamma(q).status = \text{EF}$ and $\gamma'(q).status \neq \text{EF}$. Now, because $\gamma(q).status = \text{EF}$, q can only execute **R**-action. However, as $q \in RealChildren(p)$, $KinshipOk(q,p)$ holds in γ and then q is not a root. So, q cannot execute any action and change its status during $\gamma \mapsto \gamma'$, a contradiction. ■

Theorem 4.6

Let γ be a configuration containing abnormal trees and where all abnormal trees are dead. In at most n rounds from γ, the system reaches a configuration where the status of all abnormal roots is EF*.*

Proof: Consider any execution $e = (\gamma_i)_{i \geq 0}$ starting from a configuration γ_0 that contains abnormal trees and where all abnormal trees are dead. $\forall i > 0$, we denote by γ_{R_i} the

last configuration of the i^{th} round and so the first configuration of the $(i+1)^{\text{th}}$ round. Moreover, let $\gamma_{R_0} = \gamma_0$ be the initial configuration.

Claim 1: $\forall p \in V$, $\forall i \geq R_0$, if $\gamma_i(p).status \neq$ EB, then $\forall j \geq i, \gamma_j(p).status \neq$ EB.

Proof of the claim: Assume by contradiction that $\gamma_j(p).status \neq$ EB and $\gamma_{j+1}(p).status =$ EB, with $\gamma_j \mapsto \gamma_{j+1}$. Then, $p.status =$ C in γ_j and **EB**-action is enabled at p in γ_j. So, p is in an alive abnormal tree in γ_j (Lemma 4.8), a contradiction to Lemma 4.3. □

In any configuration γ, we denote by $MaxLengthKPath(p) = \max\{|KPath(q)|, q \in V \wedge p \in KPath(q)\}$. Again in γ, we define $L(p) = MaxLengthKPath(p) - |KPath(p)|$ and $EBL(p,k) \equiv p.status =$ EB $\wedge\, L(p) = k$.

Claim 2: $\forall i \geq R_0$, if $EBL(p,k_i)$ holds in γ_i, then $\forall j \geq i, \forall k_j < k_i, \neg EBL(p,k_j)$ holds in γ_j.

Proof of the claim: If $j = i$, $EBL(p,k_j)$ is FALSE for $k_j < k_i$ because $L(p)$ cannot have two different values in a same configuration. Assume now $j > i$. The case $k_i = 0$ is direct. Assume $k_i > 0$. Assume by contradiction that $EBL(p,k_i)$ holds in γ_i and $EBL(p,k_j)$ holds in γ_j with $j > i$ and $k_j < k_i$. So, $\gamma_i(p).status = \gamma_j(p).status =$ EB and there are two cases:

- $p.status =$ EB in all the configurations between γ_i and γ_j. Consider the step $\gamma_i \mapsto \gamma_{i+1}$. Let q be any process such that $p \in KPath(q)$ in γ_i. So, $KPath(q) = q_0 \dots q_i \dots q_k$ where $q_i = p$ and $q_k = q$, and $S\text{-}Trace(KPath(q)) =$ EB$^+$EF* in γ_i. There is a unique process in $KPath(q)$ that can execute an action in $\gamma_i \mapsto \gamma_{i+1}$ (the only one of status EB with children of status EF). If it executes an action, it is **EF**-action which maintains $KinshipOk$ relation. Hence, $\forall q' \in KPath(q)$ in γ_i, $q' \in KPath(q)$ in γ_{i+1}.
  We can apply this latter property to every process r such that $p \in KPath(r)$ and $|KPath(r)| = MaxLengthKPath(p)$ in γ_i: $p \in KPath(r)$ in γ_{i+1} and the value of $|KPath(r)|$ in γ_{i+1} is greater than or equal to the value of $|KPath(r)|$ in γ_i. So, $EBL(p,k_{i+1})$ holds with $k_{i+1} \geq k_i$. Applying the same argument on step $\gamma_{i+1} \mapsto \gamma_{i+2}$, *etc.*, until step $\gamma_{j-1} \mapsto \gamma_j$, we obtain that $EBL(p,k_j)$ is TRUE in γ_j with $k_j \geq k_i$, a contradiction.
- There is a configuration between γ_i and γ_j where $p.status \neq$ EB. So, $\exists x$, $i < x < j$, such that $\gamma_x(p).status \neq$ EB and $\gamma_{x+1}(p).status =$ EB. This contradicts Claim 1.

□

We show by induction that $\forall i \geq R_d$ with $d \geq 1$, $\forall p \in V$, $\forall k \leq d-1$, $EBL(p,k)$ is FALSE in γ_i.

Base case: There are three cases:

1. If $L(p) = 0$ in γ_{R_0} and $\gamma_{R_0}(p).status =$ EB, then **EF**-action is enabled at p in γ_{R_0}, p executes **EF**-action during the first round, by Lemma 4.19 and p gets status EF. By Claim 1, $p.status$ remains different from EB forever and $EBL(p,0)$ is FALSE in γ_i, $\forall i \geq R_1$.
2. If $\gamma_{R_0}(p).status \neq$ EB, $p.status \neq$ EB forever (Claim 1) and then $EBL(p,0)$ is FALSE forever.
3. If $EBL(p,k)$ holds in γ_{R_0} with $k > 0$, $EBL(p,0)$ is FALSE forever (Claim 2).

Induction hypothesis: $\forall i \geq R_d$ with $d \geq 1$, $\forall p \in V$, $\forall k \leq d-1$, $EBL(p,k)$ is FALSE in γ_i.

Induction step: There are four cases:

1. If $L(p) = d$ and $\gamma_{R_d}(p).status = \text{EB}$, then $\forall q \in RealChildren(p)$ in γ_{R_d}, $L(q) < d$ by definition and $\gamma_{R_d(q)}.status \neq \text{EB}$ by induction hypothesis. Now, the trees are dead, so $\gamma_{R_d}(q).status = \text{EF}$. Hence, **EF**-action is enabled at p in γ_{R_d}. By Lemma 4.19, p executes **EF**-action during the round and gets status EF. Then, $p.status \neq \text{EB}$ forever (Claim 1), so $EBL(p,d)$ is FALSE at the end of the $(d+1)^{\text{th}}$ round and remains FALSE forever.
2. If $L(p) = d$ and $\gamma_{R_d}(p).status \neq \text{EB}$, then, using Claim 1, $p.status \neq \text{EB}$ forever. So, $EBL(p,d)$ is FALSE forever.
3. If $L(p) < d$, $\gamma_{R_d}(p).status \neq \text{EB}$ by induction hypothesis and we conclude as in case 2.
4. If $EBL(p,k)$ holds in γ_{R_d} with $k > d$, $EBL(p,i)$ is FALSE forever $\forall i \leq d$ (Claim 2).

With $d = n$, we have $\forall i \geq R_n$, $\forall p \in V$, $\forall k \leq n-1$, $EBL(p,k)$ is FALSE in γ_i: hence, in at most n rounds, there is no more process of status EB in abnormal trees, those ones being dead. So, all processes (and in particular the abnormal roots) in abnormal trees have status EF. ■

Lemma 4.20

If all abnormal trees are dead and **R***-action is enabled at a process p, then* **R***-action remains enabled at p until p executes it.*

Proof: Let γ be a configuration, where all abnormal trees are dead. Assume, by contradiction, that **R**-action is enabled at a process p in a configuration γ and is not enabled in the next configuration γ', but p did not execute **R**-action during the step $\gamma \mapsto \gamma'$. Notice that p does not execute any action during this step, as guards are mutually exclusive.

As **R**-action is enabled in γ and p does not execute an action during the step, $\gamma(p).status = \gamma'(p).status = EF$.

If $SelfRoot(p) \wedge \neg SelfRootOk(p)$ holds in γ, it also holds in γ' because p does not execute an action between γ and γ' and these predicates only depends on the local state of p.

Otherwise $\neg SelfRoot(p) \wedge \neg KinshipOk(p, p.par)$ holds in γ. Let $q = p.par$. If q does not execute an action between γ and γ', p is still an abnormal root. Otherwise, three cases are possible:

- $\neg GoodIdRoot(p,q)$ holds in γ.
  1. If $\gamma(p).idRoot < \gamma(q).idRoot$. If q executes **EB**-action or **EF**-action during the step, the $idRoot$ of q does not change, so $\gamma'(p).idRoot < \gamma'(q).idRoot$, and then $AbnormRoot(p)$ holds in γ'. Otherwise q executes **R**-action or **J**-action. Then $\gamma'(q).status = \text{C}$, so $\neg GoodStatus(p,q)$ and $AbnormRoot(p)$ holds in γ'.
  2. If $\gamma(p).idRoot \geq p.id$, the $idRoot$ is not modified during the step, so $\gamma'(p).idRoot = \gamma(p).idRoot \geq p.id$ and $AbnormRoot(p)$ holds in γ'.

- $\neg GoodLevel(p,q)$ holds in γ. Hence, $\gamma(p).idRoot = \gamma(q).idRoot$ but $\gamma(p).level \neq \gamma(q).level + 1$. First, if q executes **EB**-action or **EF**-action, its *idRoot* and its *level* do not change, so $\gamma'(p).idRoot = \gamma'(q).idRoot$ and $\gamma'(p).level \neq \gamma'(q).level + 1$, so $AbnormRoot(p)$ holds in γ'. Otherwise, q executes **R**-action or **J**-action and consequently $\gamma'(q).status = \text{C}$. So $\neg GoodStatus(p,q)$ and $AbnormRoot(p)$ holds in γ'.
- $\neg GoodStatus(p,q)$ holds in γ. Then $\gamma(q).status = \text{C}$, and q can only execute **EB**-action or **J**-action between γ and γ'. If q executes **EB**-action, then $EBroadcast(q)$ holds in γ, so q is in an abnormal tree (Lemma 4.8). But, by hypothesis, all abnormal trees are dead in γ, so $\gamma(q).status \neq \text{C}$, a contradiction. If q executes **J**-action then $\gamma'(q).status = \text{C}$, so $\neg GoodStatus(p,q)$ and $AbnormRoot(p)$ holds in γ'.

Thus, $\gamma'(p).status = \text{EF}$ and $AbnormRoot(p)$ holds in γ' and, consequently, $Allowed(p)$ is FALSE in γ'. So $\exists q \in p.\mathcal{N}$ such that $q \in Children(p) \wedge \neg KinshipOk(q,p)$ holds in γ' but $\gamma'(q).status = \text{C}$. Two cases are possible:

- If $q \notin Children(p)$ in γ, then q executes **J**-action during the step $\gamma \mapsto \gamma'$ and $Min(q) = p$. But $\gamma(p).status = \text{EF}$, a contradiction.
- Otherwise $q \in Children(p)$ in γ and $\gamma(q).status \neq \text{C}$. q executes either **EF**-action and $\gamma'(q).status = \text{EF}$, or **R**-action and $\gamma'(q).par \neq p$, so $q \notin Children(p)$ in γ', a contradiction. ■

Definition 4.14 (*Abnormal process*)

A process p is said to be *abnormal* if and only if p belongs to an abnormal tree. p is said to be *normal*, otherwise.

As no process can join a dead abnormal tree (Remark 4.1) and, by Lemma 4.3, no alive abnormal tree can be created, we have the following remark:

Remark 4.7

In a configuration where every abnormal tree is dead, the number of abnormal processes can only decrease.

Theorem 4.7

Starting from a configuration where every abnormal tree is dead and the status of their roots is EF*, there is no more abnormal processes in at most n rounds.*

Proof: Let γ_0 be a configuration where all abnormal trees are dead and the status of their roots is EF. By Observation 4.1, all abnormal processes have status EF in γ_0. So, from γ_0, no process can be ever an abnormal process with a status different of EF (such a process can only execute **R**-action, then it is a normal process forever, by Lemma 4.3). Then, by definition, the number of abnormal processes in γ_0 is at most n. Moreover, by Remark 4.7, it is sufficient to show that in any configuration γ_k reachable from γ_0, if the number of abnormal processes is not null, then at least one of them becomes normal within the next round.

So, let assume that some process p is abnormal in γ_k. Then, $\gamma_k(p).status = \text{EF}$. By Observation 4.1 and Lemma 4.20, the initial extremity r of $KPath(p)$ is an abnormal

process (of status EF) and executes **R**-action within the next round. After executing **R**-action, r is normal (actually, r becomes a self root), and we are done. ■

By definition, the root of a normal tree has status C. So, by Observation 4.1, we have:

Remark 4.8

Every process has status C *in a configuration containing no abnormal processes. Moreover, this configuration is clean.*

Using Lemma 4.17 and Theorems 4.5 to 4.7, we can conclude:

Theorem 4.8

In at most $3n$ *rounds, the system reaches a clean configuration.*

Then, using Theorems 4.4 and 4.8 we get:

Theorem 4.9 (*Round Complexity*)

In at most $3n + \mathcal{D}$ *rounds, the system reaches a terminal configuration.*

Lower Bound on the Worst Case Stabilization Time in Rounds. We now show that the bound proposed in Theorem 4.9 cannot be improved. To see this, we exhibit a construction that gives, $\forall n \geq 4$, $\forall \mathcal{D}, 2 \leq \mathcal{D} \leq n-2$, a network of n processes whose diameter is $\mathcal{D}$ from which there is a possible synchronous execution that lasts exactly $3n+\mathcal{D}$ rounds. (Recall that every synchronous execution is possible under the distributed unfair daemon.)

We consider a network $\mathcal{G} = (V, E)$ composed of n processes $V = \{p_1, \ldots, p_n\}$ such that p_i has ID i, for $i \in \{1, \ldots, n\}$. Figure 4.8a shows the system in its initial configuration. In details, processes p_1, p_n, ...,p_2 form a chain, *i.e.*, $\{p_1, p_n\} \in E$ and $\{p_i, p_{i-1}\} \in E$, $\forall i \in \{3, \ldots, n\}$.

We add k edges to p_2, with $2 \leq k \leq n-2$, as follows:

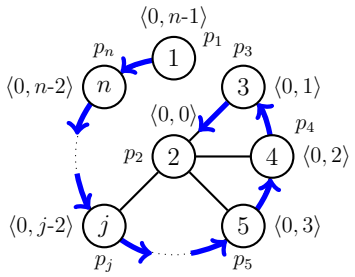
If $k = n-2$, $\{p_2, p_1\} \in E$ and for $\forall i \in \{4, \ldots, n\}$, $\{p_2, p_i\} \in E$,

Otherwise $\forall i \in \{4, \ldots, k+3\}$, $\{p_2, p_i\} \in E$.

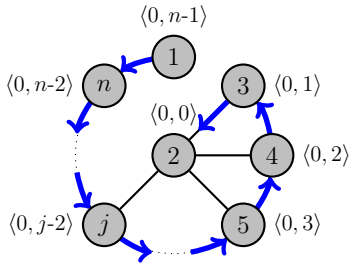
Notice that the diameter of the network is $n-k$ and can be adjusted by adding or removing some edges to p_2.

We assume the following initial configuration:

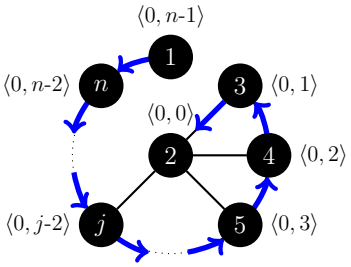
- $p_i.idRoot = 0$, $\forall i \in \{1, \ldots, n\}$,
- $p_1.level = n-1$ and $p_1.par = p_n$,
- $p_2.par = p_2$ and $p_2.level = 0$,
- $p_i.level = i-2$ and $p_i.par = p_i - 1$, $\forall i \in \{3, \ldots, n\}$.



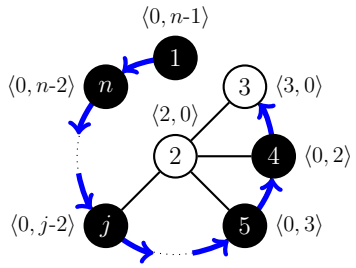
(a) The initial configuration. $\{p_2, p_j\}$ is the "last" edge to p_2.



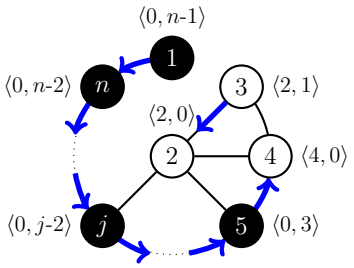
(b) In n rounds, the EB-wave reaches p_1.



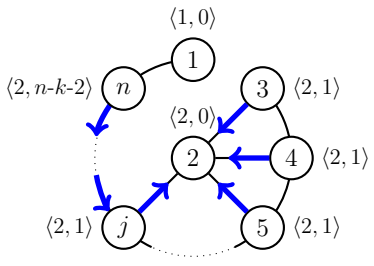
(c) In n rounds, p_2 gets status EF.



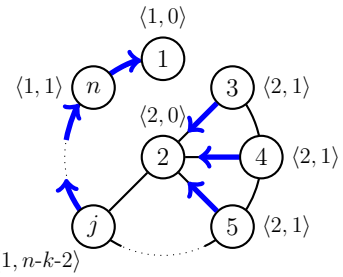
(d) p_2 and p_3 sequentially execute **R**-action.



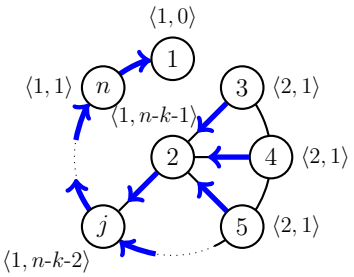
(e) p_3 executes **J**-action and p_4 simultaneously executes **R**-action.



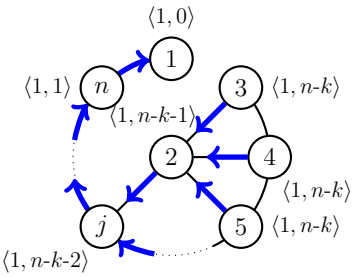
(f) In $n-3$ rounds, the cleaning is finished.



(g) In $n-k-2$ rounds, processes n to $k-3$ joins $Tree(1)$.



(h) Processes p_2 and p_{k-2} simultaneously execute **J**-action.



(i) In one round, the system reaches a terminal configuration where p_1 is the leader.

Figure 4.8 – An example in $3n + \mathcal{D}$ rounds. $(j = k + 3)$

We consider a synchronous daemon, *i.e.*, in a configuration γ, every process in $Enabled(\gamma)$ is selected by the daemon to execute an action. So, in this case, every round lasts exactly one step.

The execution is then as follows:

- $p_2, p_3, p_4, \ldots, p_n, p_1$ sequentially execute **EB**-action: n rounds. (See Figure 4.8b.)
- $p_1, p_n, p_{n-1}, \ldots, p_2$ sequentially execute **EF**-action: n rounds. (See Figure 4.8c.)
- p_2 and p_3 sequentially execute **R**-action: 2 rounds. (See Figure 4.8d.)
- For $i = 4, \ldots, n$, simultaneously p_i and p_{i-1} respectively executes **R**-action and **J**-action, in particular, p_{i-1} joins $Tree(p_2)$: $n - 3$ rounds. (See Figures 4.8e and 4.8f.)

- p_1 executes **R**-action and p_n executes **J**-action simultaneously: 1 round.
- For $i = n, \ldots, k+3$, i executes **J**-action to join $Tree(1)$: $n-k-2$ rounds. (See Figure 4.8g.)
- p_2 and p_{k+2} simultaneously execute **J**-action to join $\mathcal{T}(1)$: 1 round. (See Figure 4.8h.)
- $p_3, \ldots, p_{k+1}$ simultaneously execute **J**-action and then the configuration is terminal: 1 round. (See Figure 4.8i.)

Hence, the execution lasts exactly $3n + (n-k) = 3n + \mathcal{D}$ rounds. Using Theorem 4.9 we can conclude:

Theorem 4.10

In the worst case, the round complexity of $\mathcal{LE}$ is exactly $3n + \mathcal{D}$ rounds.

Lower Bound on the Worst Case Stabilization Time in Steps. We show that the bound given in Theorem 4.1 can be asymptotically matched, *i.e.*, we give an example of possible execution that stabilizes in $\Omega(n^3)$ steps, for every $n \geq 4$.

We consider a network $\mathcal{G} = (V, E)$ composed of n processes $V = \{p_1, \ldots, p_n\}$ such that p_i has ID $n + i$, $\forall i \in \{1, \ldots, n\}$. Figure 4.9a shows the network in this initial configuration. In details, there are $2n-3$ edges: $\{p_i, p_{i+1}\}$ for $i \in \{1, \ldots, n-2\}$ and $\{p_i, p_n\}$ for $i \in \{1, \ldots, n-1\}$. (Notice that the diameter of this network is 2.) The initial configuration is as follows:

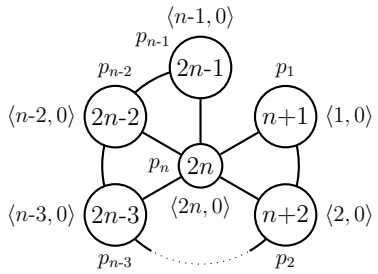
- $p_i.idRoot = i$, $\forall i \in \{1, \ldots, n-1\}$, and $p_n.idRoot = 2n$.
- $p_i.par = p_i$, $p_i.level = 0$ and $p_i.status = \text{C}$, $\forall i \in \{1, \ldots, n\}$.

We consider the following execution:

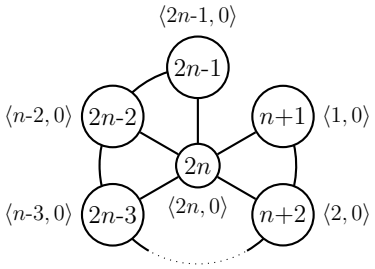
- For $i = n-1, n-2, \ldots, 1$, we clean $Tree(p_i)$ the following way:
  1. If $i \leq n-2$, for $j = n-2, n-1, \ldots, i$,
     a) For $k = j+1, j+2, \ldots, n-1$, p_k joins $Tree(p_j)$.

     Case 1 lasts $\sum_{j=1}^{n-1-i} j = (n-i-1)(n-i)/2$ steps.
  2. $p_i, p_{i+1}, \ldots, p_{n-1}$ sequentially execute **EB**-action: $n-i$ steps.
  3. $p_{n-1}, p_{n-2}, \ldots, p_i$ sequentially execute **EF**-action: $n-i$ steps.
  4. $p_i, p_{i+1}, \ldots, p_{n-1}$ sequentially execute **R**-action: $n-i$ steps.

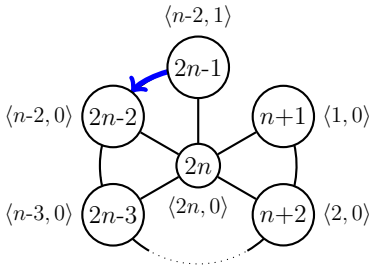
  Figures 4.9e to 4.9h show the cleaning of $Tree(p_{n-3})$.
- After all abnormal trees have been cleaned, processes p_{n-1} to p_2 join $Tree(p_1)$ similarly as Case 1: $\sum_{j=1}^{n-2} j = (n-1)(n-2)/2$ steps (Figure 4.9j).
- p_n executes **J**-action to join $Tree(p_1)$: 1 step (Figure 4.9k).



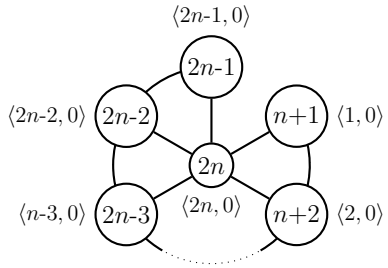
(a) The initial configuration.



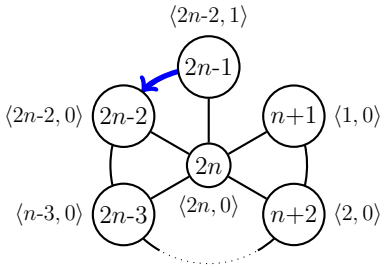
(b) In three steps, p_{n-1} becomes normal.



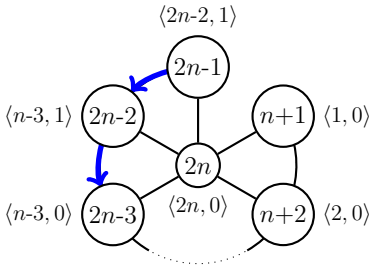
(c) p_{n-1} executes **J**-action and joins $Tree(p_{n-2})$.



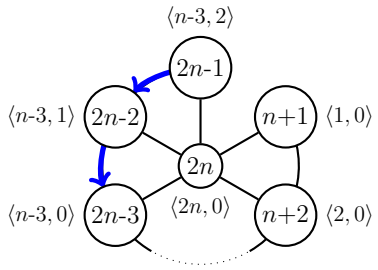
(d) In six steps, the abnormal tree rooted in p_{n-2} is cleaned.



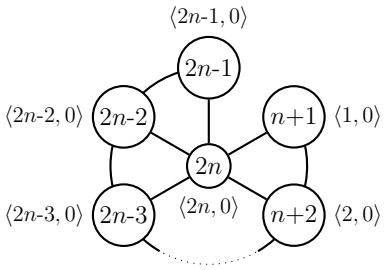
(e) p_{n-1} executes **J**-action and joins the normal tree $Tree(p_{n-2})$.



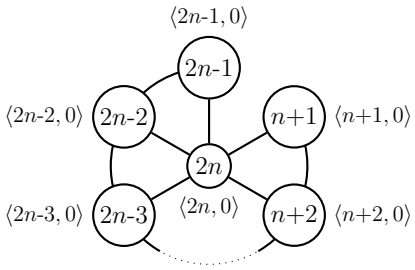
(f) p_{n-2} executes **J**-action and joins the abnormal tree $Tree(p_{n-3})$



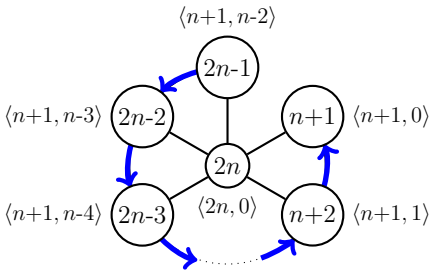
(g) p_{n-1} executes **J**-action and updates its *idRoot* to $n-3$.



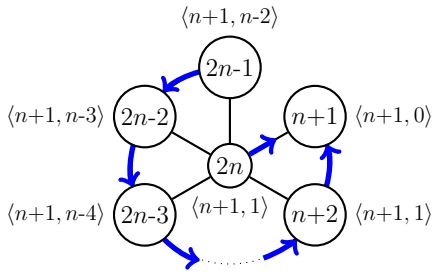
(h) In nine steps, the abnormal tree rooted in p_{n-3} is cleaned.



(i) There is no more abnormal trees.



(j) In $\sum_{j=1}^{n-2} j$ steps, processes p_{n-1} to p_2 elect p_1.



(k) In one step, the system reaches a terminal configuration where p_1 is leader.

Figure 4.9 – An example in $\Omega(n^3)$ steps.

Hence, the complete execution lasts:

$$3+\sum_{i=1}^{n-2}\left(3(n-i)+\frac{(n-i-1)(n-i)}{2}\right)+\frac{(n-1)(n-2)}{2}+1=\frac{n^3}{6}+\frac{3}{2}n^2-\frac{8}{3}n+2 \text{ steps}$$

So, there exists an execution in $\Omega(n^3)$. Using Theorem 4.3, we can conclude:

Theorem 4.11

In the worst case, the step complexity of $\mathcal{LE}$ is in $\Theta(n^3)$ steps.

4.4 Step Complexity of Algorithm $\mathcal{DLV}_1$

In this section, we study the step complexity of the algorithm given in [DLV11a], called here $\mathcal{DLV}_1$.[1] Below, we show that its stabilization time is not polynomial in steps.

First, we give the code of algorithm $\mathcal{DLV}_1$ and an informal explanation of its main principles in Subsection 4.4.1. Then, we give in Subsection 4.4.2 an example of a class of network in which there is a possible execution that stabilizes in $\Omega(2^n)$ steps.

4.4.1 Overview of $\mathcal{DLV}_1$

The formal code of Algorithm $\mathcal{DLV}_1$ is given in Algorithm 5.[2] This algorithm uses priorities. Each of its actions is given with priority number. When an enabled process is selected by the daemon, it only executes its enabled action with the lowest priority number.

Algorithm $\mathcal{DLV}_1$ elects the process of minimum ID, ℓ, and builds a breadth-first spanning tree rooted at ℓ. IDs are assumed to be natural integers. To ensure that every process knows which one is elected, it maintains a variable *leader* in which is saved the alleged leader. The level of the process in the tree is saved into variable *level*. The *key* of a process p is the combination of its two variables $p.leader$ and $p.level$. The keys are ordered using the lexical order. Notice that there is no explicit pointer to the parent but it can easily be computed with the keys.

Notice that we suppose every ID to be different than 0. When there is a smaller possible key in the neighborhood of a process p, p may execute **A2**-action and update its key accordingly.

As in $\mathcal{LE}$, a "good relation" between a process p and its parent, called $Valid(p)$, is defined. This predicate ensures that p is either a self root ($\langle p, 0\rangle$), a zero root ($\langle 0, 0\rangle$), or its key is greater or equal to the best possible key.

Zero propagation. The main difference between $\mathcal{LE}$ and $\mathcal{DLV}_1$ is the way to deal with fake IDs. $\mathcal{DLV}_1$ exploits the value 0, smaller than any ID. More precisely, if a process p

[1] $\mathcal{DLV}_1$ stands for "Datta, Larmore, and Vemula."

[2] The code given in Algorithm 5 is slightly different from the one given in [DLV11a]. Actually, we found a flaw in the definition of the $Valid$ predicate. After private communication with the authors, we agree on the solution proposed here.

Algorithm 5 – Actions of Process p in Algorithm $\mathcal{DLV}_1$ [DLV11a].

Inputs.

- $p.id \in \mathbb{N}$
- $p.\mathcal{N}$

Variables.

- $p.leader \in \mathbb{N}$
- $p.level \in \mathbb{N}$
- $p.key = \langle p.leader, p.level \rangle$

Functions.

$$Successor(\langle lead, lvl \rangle) \equiv \langle lead, lvl + 1 \rangle$$
$$MinKeyNeighbor(p) \equiv \min \{q.key : q \in p.\mathcal{N}\}$$

Predicates.

$$SelfRoot(p) \equiv p.key = \langle p, 0 \rangle$$
$$ZeroRoot(p) \equiv p.key = \langle 0, 0 \rangle$$
$$Valid(p) \equiv SelfRoot(p) \vee ZeroRoot(p) \vee (p.key > MinKeyNeighbor(p))$$
$$Is\_Linked(p) \equiv p.key = Successor(MinKeyNeighbor(p))$$
$$Is\_Good(p) \equiv Is\_Linked(p) \vee (SelfRoot(p) \Rightarrow p.key < MinKeyNeighbor(p)) \vee ZeroRoot(p)$$
$$Frozen(p) \equiv SelfRoot(p) \wedge (\exists q \in p.\mathcal{N}, q.leader = 0)$$
$$ZeroLeaf(p) \equiv p.leader = 0 \wedge (\forall q \in p.\mathcal{N}, (q.key \leq p.key) \vee SelfRoot(q))$$

Actions.

A1 (prio. 1) :: $\neg Valid(p) \rightarrow$ **if** $p.leader < p.id$ **then** $p.key := \langle 0, 0 \rangle$ **else** $p.key := \langle p, 0 \rangle$

A2 (prio. 2) :: $\neg Is\_Good(p) \wedge \neg Frozen(p) \rightarrow p.key := Successor(MinKeyNeighbor(p))$

A3 (prio. 3) :: $ZeroLeaf(p) \rightarrow p.key := \langle p, 0 \rangle$

is not valid and if its leader is smaller than its own ID, *i.e.*, maybe a fake ID, p executes **A1**-action and gets key $\langle 0, 0 \rangle$. 0 is then propagated in the network using **A2**-action and erase any fake ID. The only processes that can make a barrier to the propagation of 0 are the self roots. Indeed a self root neighbor with a process of leader 0 is said frozen, *i.e.*, it cannot execute **A2**-action and get 0 as leader too.

When the growing of zero trees ends, the leaves, *i.e.*, processes of leader 0 that are surrounded by self roots or processes with smaller key, can reset to self root executing **A3**-action.

Example of execution. Figure 4.10 shows an example of execution of $\mathcal{DLV}_1$. For an easy reading of the figure, we explicit the parent pointers. The colors of processes are used to differentiate the leader. If the node is gray, its leader is an existing ID (we can infer which one with the parent pointers). If the node is black, its leader is the fake ID 1, smaller than any ID in the network. If the node is white, its leader is 0. We can also infer the level with the parent pointers.

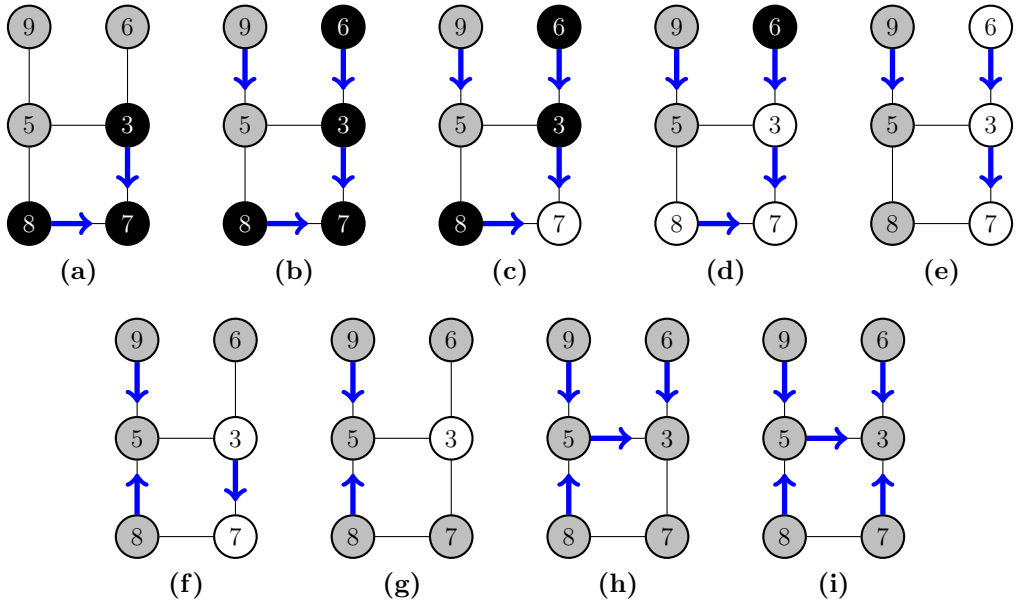


Figure 4.10 – Example of execution of algorithm $\mathcal{DLV}_1$. Arrows explicit parent pointers. The leader of white nodes (respectively, black nodes) is 0 (respectively, the fake ID 1, smaller than any ID in the network). Otherwise, the node is gray and its leader is an existing ID that can be infered using parent pointers.

In the initial configuration (Configuration (a)), the leader of processes 3, 7, and 8 is the fake ID 1. Then, in step (a) $\mapsto$ (b), 1 is propagated to process 6 that executes A_2. At the same time, 9 also executes **A2**-action and chooses 5 as leader. 7 corrects its error and becomes a zero root by executing **A1**-action during step (b) $\mapsto$ (c). The special ID 0 is propagated to processes 3 and 8 in step (c) $\mapsto$ (d) and then to process 6 in step (d) $\mapsto$ (e). At the same time, 8 can reset itself executing **A3**-action because it is a zero leaf. Notice that 0 is not propagated to 5 since 5 is a self root and cannot execute **A2**-action. In step (e) $\mapsto$ (f), 6 resets itself and 8 executes **A2**-action to choose 5 as leader. Then, 3 executes **A3**-action during step (f) $\mapsto$ (g). The last process of leader 0, process 7, resets itself during step (g) $\mapsto$ (h). In the same step, 5 and 6 execute **A2**-action and choose 3 as leader. Notice that the leader of 8 and 9 is still 5 in Configuration (h). Leader 3 is propagated to 7, 8, and 9 during step (h) $\mapsto$ (i). Hence, 3 is elected in Configuration (i).

4.4.2 Example of Exponential Execution

In this subsection, we propose a class of network of n processes, for every $n \geq 5$, in which there exists an execution of $\mathcal{DLV}_1$ in $\Omega(2^n)$ steps.

Network and initial configuration. We consider a network composed of $n \geq 5$ processes p_k of ID $k \in \{2, \ldots, n+1\}$. Notice that 1 is a fake ID smaller than every ID in the network. Figure 4.11 shows the network and the initial configuration. The network is composed of $H = \left\lfloor \frac{n-1}{4} \right\rfloor$ *diamonds*. $\forall h \in \{0, \ldots, H-1\}$, *Diamond* h is made of the following edges: $\{p_{4(H-h-1)+2}, p_{4(H-h-1)+3}\}$, $\{p_{4(H-h-1)+2}, p_{4(H-h-1)+4}\}$, $\{p_{4(H-h-1)+3}, p_{4(H-h-1)+5}\}$, $\{p_{4(H-h-1)+4}, p_{4(H-h-1)+6}\}$, and $\{p_{4(H-h-1)+5}, p_{4(H-h-1)+6}\}$.

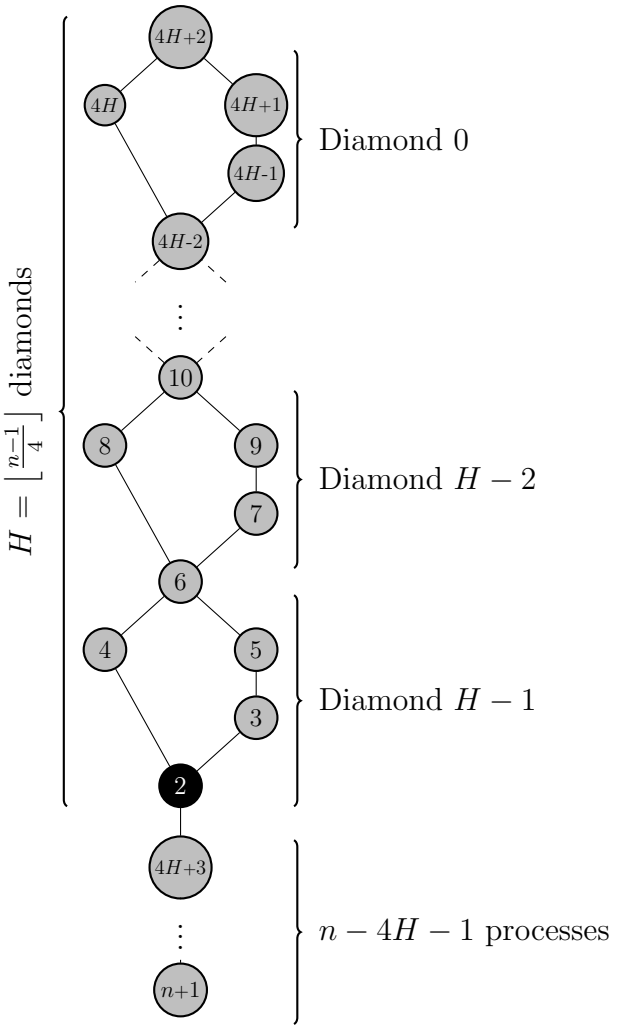


Figure 4.11 – Initial configuration for any $n \geq 5$.

The remaining processes form a chain linked to p_2, *i.e.*, the edges $\{p_i, p_{i+1}\}$ with $i \in \{4H+4, n\}$, and the edge $\{p_2, p_{4H+3}\}$.

We consider the initial configuration where $p_2.key = \langle 1, 0\rangle$, *i.e.*$p_2$ has the fake id 1 as leader, and $\forall i \in \{3, \ldots, n+1\}$, $p_i.key = \langle i, 0\rangle$, *i.e.*, p_i is self root.

Overview of the execution for $n = 11$. Figure 4.12 shows an intuitive idea of the execution for $n = 11$. Each phase is composed of three waves: the propagation of fake ID 1, the propagation of special ID 0, and the reset.

During the first phase ((1) in Figure 4.12), the fake ID 1 is propagated to p_4, p_6, p_8, and p_{10}. The fake ID 1 is also propagated to p_3 and p_7 to prepare the next phases. Then, p_2 corrects its error executing **A1**-action and the special ID 0 is propagated along the same path. The reset starts at p_{10}, and then p_8 resets.

p_7 still has 1 as leader so, in phase (2), 1 is propagated to p_9 and p_{10}. Then, since p_6 holds 0, 0 propagated to p_7, p_9, and p_{10}. Finally, p_{10}, p_9, p_7, p_6 and p_4 resets.

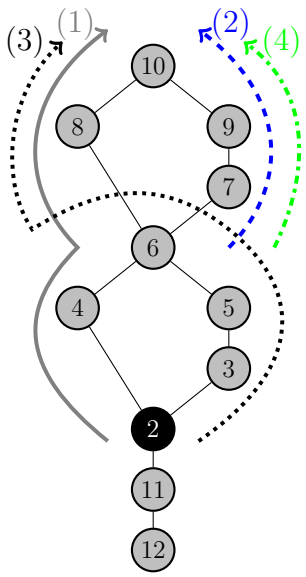


Figure 4.12 – Intuitive idea of the execution for $n = 11$.

Then, during phase (3), we start again on the right side. $p_3.leader = 1$ so 1 is propagated to p_5, p_6, p_8, and p_{10}. Then 0 is propagated to p_3 and along the same path. The reset starts from p_{10} to p_8 as in phase (1).

Finally phase (4) is similar to phase (2) with a reset along the right side of the network.

Notice that the additional processes p_{11} and p_{12} do nothing.

Generalization for any $n \geq 5$. We generalize this idea for any $n \geq 5$. We consider an unfair daemon that selects the enabled processes according to function *Daemon* given in Algorithm 6.

Theorem 4.12

For every $n \geq 5$, there exists a network of n processes in which there exists a possible execution of Algorithm $\mathcal{DLV}_1$ that stabilizes in $\Omega(17 \times (2^{\lfloor \frac{n-1}{4} \rfloor} - 1)$ steps.

Proof: Let consider the diamond h. When $p_{4(H-h-1)+2}$ holds 1 as leader, the processes into diamond h executes the following actions:

- Propagation of 1 on the left: $p_{4(H-h-1)+3}$, $p_{4(H-h-1)+4}$, $p_{4(H-h-1)+6}$ executes **A2**-action
- Propagation of 0 on the left: $p_{4(H-h-1)+2}$ executes Action A_1, $p_{4(H-h-1)+4}$, $p_{4(H-h-1)+6}$ executes **A2**-action
- Reset on the left: $p_{4(H-h-1)+6}$, $p_{4(H-h-1)+4}$ executes **A3**-action
- Propagation of 1 on the right: $p_{4(H-h-1)+5}$, $p_{4(H-h-1)+6}$ executes **A2**-action
- Propagation of 0 on the right: $p_{4(H-h-1)+3}$, $p_{4(H-h-1)+5}$, $p_{4(H-h-1)+6}$ executes **A2**-action
- Reset on the right: $p_{4(H-h-1)+6}$, $p_{4(H-h-1)+5}$, $p_{4(H-h-1)+3}$, $p_{4(H-h-1)+2}$

Algorithm 6 – Algorithm of the daemon.

```
1: function Daemon
2: |   p_3 executes A2-action;
3: |   BuildLeft(H − 1, 1);
4: |   p_2 executes A1-action;
5: |   BuildLeft(H − 1, 0);
6: |   ResetLeft(H − 1);

7: function BuildLeft(h, b)
8: |   p_{4(H−h−1)+4} executes A2-action;
9: |   p_{4(H−h−1)+6} executes A2-action;
10: |   if h > 0 then
11: |   |   if b = 1 then
12: |   |   |   p_{4(H−h−1)+7} executes A2-action;
13: |   |   BuildLeft(h − 1, b);

14: function BuildRight(h)
15: |   if b ≠ 1 then
16: |   |   p_{4(H−h−1)+3} executes A2-action;
17: |   p_{4(H−h−1)+5} executes A2-action;
18: |   p_{4(H−h−1)+6} executes A2-action;
19: |   if h > 0 then
20: |   |   if b = 1 then
21: |   |   |   p_{4(H−h−1)+7} executes A2-action;
22: |   |   BuildLeft(h − 1, b);

23: function ResetRight(h)
24: |   if h = 0 then
25: |   |   p_{4∗(H−1)+6} executes A3-action;
26: |   else
27: |   |   ResetLeft(h − 1);
28: |   p_{4(H−h−1)+4} executes A3-action;
29: |   BuildRight(h, 1);
30: |   BuildRight(h, 0);
31: |   ResetRight(h);

32: function ResetRight(h)
33: |   if h = 0 then
34: |   |   p_{4∗(H−1)+6} executes A3-action;
35: |   else
36: |   |   ResetLeft(h − 1);
37: |   p_{4(H−h−1)+5} executes A3-action;
38: |   p_{4(H−h−1)+3} executes A3-action;
39: |   p_{4(H−h−1)+2} executes A3-action;
```

So we have 17 actions. Notice that $p_{4(H-h-1)+6}$ holds 1 as leader twice. Hence, if $h \geq 1$, one such execution on diamond h implies two executions on diamond $h-1$. We denote $T(h)$ the maximum number of actions executed by processes on diamonds h to 0. So $T(h) \geq 17 + 2T(h-1)$, for $h \geq 1$. Notice that this execution on diamond 0 does not imply any other actions, so $T(0) \geq 17$.

We can trivially prove by induction that $T(h) \geq 17 \sum_{i=0}^{h} 2^i$. Hence,

$$T(H-1) \geq 17 \sum_{i=0}^{H-1} 2^i = 17(2^H - 1) = 17(2^{\lfloor \frac{n-1}{4} \rfloor} - 1)$$

■

4.5 Step Complexity of Algorithm $\mathcal{DLV}_2$

In this section, we study the step complexity of the algorithm given in [DLV11b], called here $\mathcal{DLV}_2$. Just as for $\mathcal{DLV}_1$, we show that its stabilization time is not polynomial in steps.

First, we give the code of algorithm $\mathcal{DLV}_2$ and an informal explanation of its main principles in Subsection 4.5.1. Then, we give in Subsection 4.5.2 an example of a class of network in which there is a possible execution that stabilizes in $\Omega(n^4)$ steps. Finally, in Subsection 4.5.3, we generalize the previous example to a class of network where there is a possible execution that stabilizes in $\Omega(n^\alpha)$ for any $\alpha \geq 4$.

4.5.1 Overview of $\mathcal{DLV}_2$

The formal code of Algorithm $\mathcal{DLV}_2$ is given in Algorithm 7.

The principle of Algorithm $\mathcal{DLV}_2$ is very similar to Algorithm $\mathcal{DLV}_1$. It elects ℓ and builds a breadth-first search spanning tree rooted at ℓ. A variable *leader* is used to save the ID of the current leader. Variables *par* and *level* are used to define the tree. The *key* of a process p is the combination of its two variables $p.leader$ and $p.level$. Notice that the keys are ordered using the classical lexical order.

Let p be a process. Let q be its neighbor of smallest key ($BestNbrKey(p)$). Suppose the key of process p is not the immediate successor of the q's key or $p.par \neq q$. p may execute **J**-action to modify its *key* and its *par* pointer accordingly. Notice that, contrary to our algorithm, p can execute **J**-action and change its parent when there is a neighbor with the same leader but with a level smaller than $p.level - 1$, in order to build a breadth-first spanning tree. Note also that the execution of **J**-action is constrained by the use of a *color*, whose goal will be explained later.

As in $\mathcal{LE}$ and $\mathcal{DLV}_1$, Datta et al. define a "good relation" between a process p and its parent called $IsTrueChld(p)$. This ensures that the key of p is the successor key of its parent and that its leader is smaller than its own ID. Then, a maximal set of processes linked by *par* pointers and satisfying the $IsTrueChld$ relation defines a tree. The root of a tree can be a *true root* ($IsTrueRoot(p)$), *i.e.*, the key of p is a self key ($\langle p, 0 \rangle$). In this case, the tree is said to be *normal*. Otherwise, the root is a *false root* ($IsFalseRoot(p)$), *i.e.*, neither a true root nor a true child, and the tree is said to be *abnormal*.

Algorithm 7 – Actions of Process p in Algorithm $\mathcal{DLV}_2$ [DLV11b].

Inputs.

- $p.id \in \mathbb{N}$
- $p.\mathcal{N}$

Variables.

- $p.leader \in \mathbb{N}$
- $p.level \in \mathbb{N}$
- $p.key = \langle p.leader, p.level \rangle$
- $p.par \in p.\mathcal{N} \cup \{p\}$
- $p.color \in \{1, 2\}$
- $p.done \in \mathbb{B}$

Functions.

$$
\begin{array}{lcl}
SelfKey(p) & \equiv & \langle p.id, 0 \rangle \\
SuccKey(p) & \equiv & \langle p.leader, p.level + 1 \rangle \\
BestNbrKey(p) & \equiv & \min \{q.key : q \in p.\mathcal{N} \wedge SuccKey(q) < SelfKey(p) \wedge q.color = 2\} \\
TrueChldrn(p) & \equiv & \{q \in p.\mathcal{N} : q.par = p \wedge q.key = SuccKey(p)\} \\
FalseChldrn(p) & \equiv & \{q \in p.\mathcal{N} : q.par = p \wedge q.key \neq SuccKey(p)\} \\
Recruits(p) & \equiv & \{q \in p.\mathcal{N} : q.key > SuccKey(p)\}
\end{array}
$$

Predicates.

$$
\begin{array}{lcl}
IsTrueRoot(p) & \equiv & p.key = SelfKey(p) \\
IsTrueChld(p) & \equiv & p.key = SuccKey(p.par) \wedge q.key > p.id \\
IsFalseRoot(p) & \equiv & \neg IsTrueRoot(p) \wedge \neg IsTrueChld(p) \\
Done(p) & \equiv & Recruits(p) = \emptyset \wedge (\forall q \in TrueChldrn(p), q.done) \\
ColorFrozen(p) & \equiv & IsTrueRoot(p) \wedge p.done
\end{array}
$$

Guards.

$$
\begin{array}{lcl}
Join(p, q) & \equiv & IsFalseRoot(p) \vee SuccKey(q) < p.key \wedge q.color = 2 \\
 & & \quad \wedge\ q.key = BestNbrKey(p) \wedge FalseChldrn(p) = \emptyset \\
Reset(p) & \equiv & IsFalseRoot(p) \\
Color1(p) & \equiv & p.color = 2 \wedge \neg ColorFrozen(p) \wedge p.par.color = 2 \\
 & & \quad \wedge\ Recruits(p) = \emptyset \wedge (\forall q \in TrueChldrn(p), q.color = 1) \\
Color2(p) & \equiv & p.color = 1 \wedge \neg ColorFrozen(p) \wedge p.par.color = 1 \\
 & & \quad \wedge\ (\forall q \in TrueChldrn(p), q.color = 2) \\
UpdateDone(p) & \equiv & p.done \neq Done(p)
\end{array}
$$

Actions.

J	(prio. 1)	::	$\exists q \in p.\mathcal{N}, Join(p, q)$	$\rightarrow$	$p.key := SuccKey(q)$ $p.par := q$ $p.color := 1$ $p.done :=$ False
R	(prio. 2)	::	$Reset(p)$	$\rightarrow$	$p.key := SelfKey(p)$ $p.par := p$ $p.color := 2$ $p.done :=$ False
C1	(prio. 3)	::	$Color1(p)$	$\rightarrow$	$p.color := 1$ $p.done := Done(p)$
C2	(prio. 3)	::	$Color2(p)$	$\rightarrow$	$p.color := 2$ $p.done := Done(p)$
UD	(prio. 4)	::	$UpdateDone(p)$	$\rightarrow$	$p.done := Done(p)$

(a) 7 can execute **C2**-action and get color 2. **(b)** 7 can execute **C1**-action and get color 1.

Figure 4.13 – Guards of color actions. The ID is represented inside the node. The label next to a node shows its *key*. The arrows represent *par* pointers. No arrow exits a node if its parent is itself. The filling represents the color: gray for 1 and white for 2.

Color waves. The main difference between $\mathcal{DLV}_1$ and $\mathcal{DLV}_2$ is the way to deal with these abnormal trees. Instead of using a status and a three-waves cleaning, $\mathcal{DLV}_2$ uses color waves. More precisely, each process has a variable *color*, whose value is either 1 or 2. A process can only choose as parent a neighbor of color 2 and after executing **J**-action, the process gets color 1.

A process can change its color, by executing **C1** or **C2**-action, if it has the same color than its parent (it is trivially satisfied for every true root) and if all of its true children have the other color (see Figure 4.13). There is an additional constraint to change a color to 1: as a process cannot recruit when it has color 1, a process p of color 2 must not change its color while it can recruit processes (while $Recruits(p) \neq \emptyset$).

To add a new level in the tree, the leaves must change their color to 2. Then, the goal is to propagate up in the tree a first wave of **C1**-actions initiated by the parents of the leaves, so that a second wave of **C2**-actions can be initiated by the leaves. To ensure that, the root should absorbed a (previous) wave. But, only a true root can absorb a color wave. Indeed, the priorities on actions prevent a false root to change its color (before it resets) and, so, to absorb a color wave.

Therefore, the colors of the processes in an abnormal tree eventually alternate, *i.e.*, the parents and their true children do not have the same color, and no more process can join the tree: the tree is *color locked*. Then, the false root eventually resets by executing **R**-action, and so forth. Once all abnormal trees have been removed, ℓ is a true root and regularly absorb color waves allowing then the leaves of its tree to recruit processes.

Finally, in $O(n)$ rounds, ℓ is elected and a breadth-first spanning tree rooted at ℓ is built. Notice that the color waves might never end. So, an additional mechanism allow to ensure the silence by using a Boolean variable *done* and **UD**-action. When a process p believes that the construction of the final tree is finished (because it cannot recruit processes anymore) and all its true children q (if any) have set their variables $q.done$ to True, $p.done$ is set to True. Moreover, a true root r cannot change its color once $r.done$ holds. In this case, we said that r is *color frozen*. Thus, after the completion of the final tree construction, the value True is propagated up in the tree into the *done* variables, and in $O(\mathcal{D})$ rounds, the system reaches a terminal configuration.

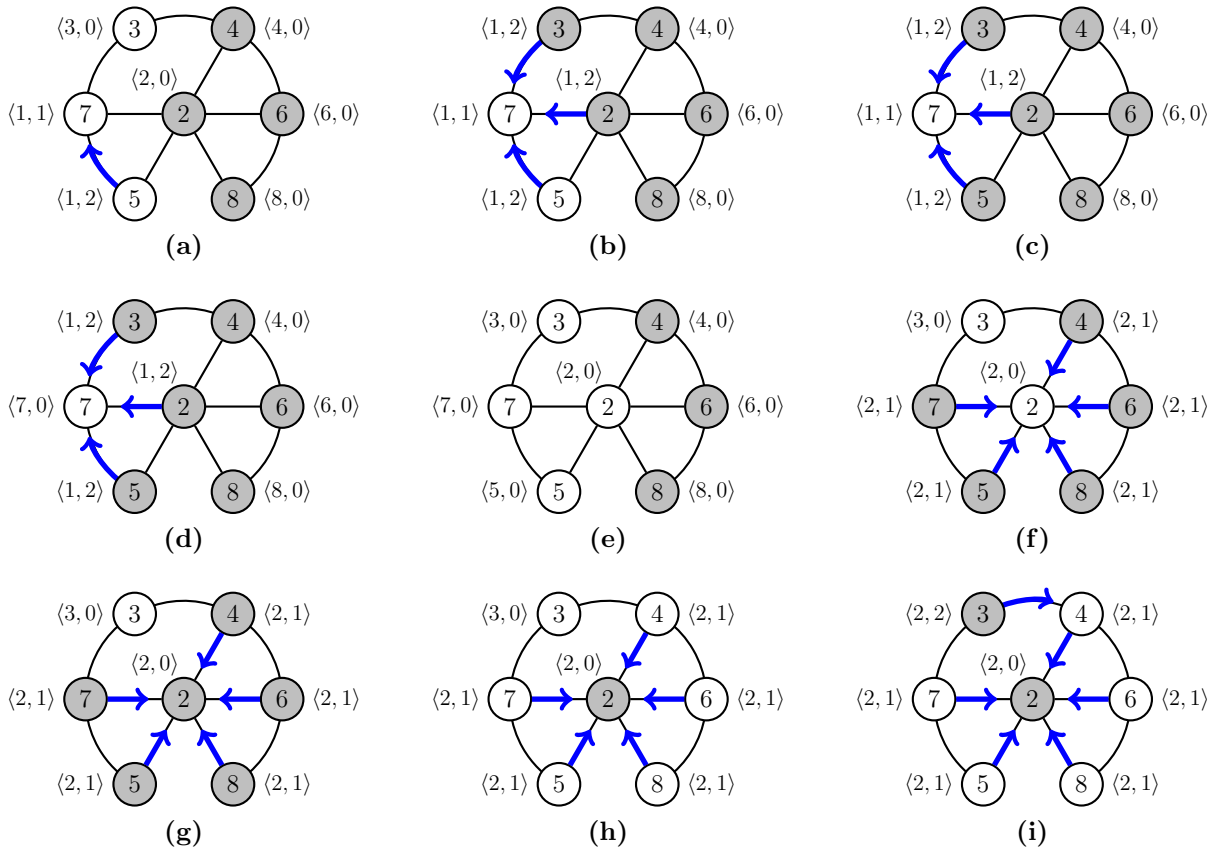


Figure 4.14 – Example of execution of $\mathcal{DLV}_2$.

Example of execution. Figure 4.14 shows an example of execution of $\mathcal{DLV}_2$ (for sake of simplicity, we do not consider the *done* variables and **UD**-actions in this example).

In the initial configuration (Configuration (a)), the leader of process 7 is 1, the only fake id. Moreover, 5 has already chosen 7 as parent. Then, in step (a) $\mapsto$ (b), 2 and 3 execute **J**-action and choose 7 (of color 2) as parent. Note also that 5 has the same color than its parent (7), has no true child, and cannot recruit any other process. So 5 executes **C1**-action and gets color 1 in (b) $\mapsto$ (c). No more process can join the tree rooted at 7 and the tree is color locked (7 is a false root and cannot change its color), so 7 resets during (c) $\mapsto$ (d). In Configuration (d), 2, 3 and 5 are false roots. In (d) $\mapsto$ (e) they execute **R**-action in turns. Then, in (e) $\mapsto$ (f), processes 4, 5, 6, 7, and 8 execute **J**-action to choose 2 as parent. In Configuration (f), 3 cannot join the tree rooted at 2 because all its neighbors have color 1. 2 changes its color to 1 by executing **C1**-action in (f) $\mapsto$ (g). Then, processes 4, 5, 6, 7, and 8 get color 2 by executing **C2**-action in (g) $\mapsto$ (h). Finally, 3 is allowed to execute **J**-action and joins the tree rooted at 2 in (h) $\mapsto$ (i).

4.5.2 Example of Execution in $\Omega(n^4)$ Steps

First, we show an execution of $\mathcal{DLV}_2$ that lasts $\Omega(n^4)$ steps.

Network and Initial Configuration. We consider a network made of $n = L \times \beta$ processes with $L = 8$ and $\beta \geq 2$: $p_{(1,1)}, p_{(1,2)}, \ldots, p_{(1,\beta)}$, $p_{(2,1)}$, $p_{(2,2)}, \ldots, p_{(2,\beta)}$, $\ldots$, $p_{(8,1)}, p_{(8,2)}, \ldots, p_{(8,\beta)}$ such that the ID of $p_{(i,j)}$ is $(i-1)\beta + j, \forall i \in \{1, \ldots, 8\}, \forall j \in \{1, \ldots, \beta\}$. Notice that 0 is a fake ID smaller than every ID in the network.

Figure 4.15a shows the structure of the network and the initial configuration. In details, the processes form β columns: $\forall i \in \{2, \ldots, 8\}, \forall j \in \{1, \ldots, \beta\}, \{p_{(i-1,j)}, p_{(i,j)}\} \in E$.

There are also three complete bipartite subgraphs: $\forall j, j' \in \{1, \ldots, \beta\}, j' \neq j$,

$$\{p_{(4,j)}, p_{(5,j')}\} \in V, \{p_{(6,j)}, p_{(7,j')}\} \in E \text{ and } \{p_{(7,j)}, p_{(8,j')}\} \in E$$

These bipartite subgraphs split the network in four layers:

- Layer 1: line 8
- Layer 2: line 7
- Layer 3: lines 5 and 6
- Layer 4: lines 1 to 4

We choose the following initial configuration.

- For $i \in \{1, \ldots, 8\}$, for $j \in \{1, \ldots, \beta\}$, $p_{(i,j)}.leader = 0$, $p_{(i,j)}.level = i$ and $p_{(i,j)}.done = \text{FALSE}$
- For $j \in \{1, \ldots, \beta\}$,
  - $p_{(1,j)}.par = p_{(1,j)}$
  - $p_{(5,j)}.par = p_{(4,1)}$
  - $p_{(7,j)}.par = p_{(6,1)}$
  - $p_{(8,j)}.par = p_{(7,1)}$
  - For $i \in \{2, 3, 4, 6\}$, $p_{(i,j)}.par = p_{(i-1,j)}$
- For $i \in \{1, \ldots, 8\}$, $p_{(i,1)}.color = (i \bmod 2) + 1$
- For $j \in \{2, \ldots, \beta\}$,
  - $p_{(8,j)}.color = 1$
  - For $i \in \{1, \ldots, 7\}$, $p_{(i,j)}.color = 2$

Overview of the execution. We first give an illustrative execution to understand the $\Omega(n^4)$ lower bound.

We start with Configuration (a) of Figure 4.15. Starting from this configuration, all the processes of the first column and of the last line successively reset. We obtain configuration (b). This costs at least β steps (since the reset of the last line can be sequential). Then, all processes $p_{(8,.)}$ can join $p_{(7,2)}$ (which has the fake id 0 as *leader*). This leads to Configuration (c). Then, we can reset $p_{(7,2)}$ and the last line (at least β

steps). Again processes $p_{(8..)}$ can join $p_{(7,3)}$ and we can reset, *etc.*, until we reset $p_{(7,\beta)}$ and the last line to obtain Configuration (d). Overall, this costs at least β^2 steps.

From Configuration (d), we can rebuilt the tree on $p_{(6,2)}$. The tree is shown in Configuration (e), and we can reset the processes following the order given by the arrow in Configuration (e). We obtain Configuration (f). Again we can start the succession of buildings and resets bottom-up just as before, but this time, until resetting a tree rooted at $p_{(5,\beta)}$ (Configuration (g)). This costs at least β^3 steps.

From Configuration (g), we can rebuild a tree on the second column until reaching Configuration (h) . This latter is similar to the first one, Configuration (a). The only difference is that the main tree is now rooted at $p_{(1,2)}$ instead of $p_{(1,1)}$. We can repeat the same scheme on each column. This leads to an execution of at least β^4 steps.

Details of the Execution. Now, let see the details of the execution. We consider an unfair daemon which selects the enabled processes according to the function *Daemon* given in Algorithm 8. In this algorithm, $top(i)$ (respectively $bottom(i)$) is the number of the first line (respectively last line) of layer i. More precisely:

$$top(i) = L - 2^{i-1} + 1$$

$$bottom(i) = \begin{cases} top(1) & \text{if } i = 1 \\ top(i-1) - 1 & \text{if } i > 1 \end{cases}$$

In $Build(layer, col)$, all the processes of lines $top(layer)$ to 8 execute line by line **J**-action. Notice that every process of line $top(layer)$ chooses the process $p_{(top(layer)-1,col)}$ as parent.

In $Reset(layer, col)$, all the processes on column col from the one on line $top(layer+1)$ to the one on line $bottom(layer+1)$ execute **R**-action (except for layer 1 where all the processes of line 8 also execute **R**-action). Then, $Reset(layer-1, i)$ and $Build(layer-1, i+1)$ are called for each col $i = 1, \ldots, \beta - 1$. Finally, $Reset(layer-1, \beta)$ is executed.

We count how many times processes $p_{(8,.)}$ executes **R**-action:

- Each process $p_{(8,.)}$ executes once **R**-action in $Reset(layer, col)$, when $layer = 1$ (line 11 of Algorithm 8): at least β processes execute **R**-action.
- $Reset(3, col)$ is called β times by function *Daemon*.
- $Reset(2, col)$ is called β times by function $Reset(3, col)$.
- $Reset(1, col)$ is called β times by function $Reset(2, col)$.

Hence, **R**-action is executed β^4 times by the processes of line 8. Now, $\beta = \frac{n}{8}$. Hence we can conclude:

Theorem 4.13

For every $\beta \geq 2$, there exists a network of $n = 8 \times \beta$ processes in which there exists a possible execution of Algorithm $\mathcal{DLV}_2$ that stabilizes in $\Omega(n^4)$ steps.

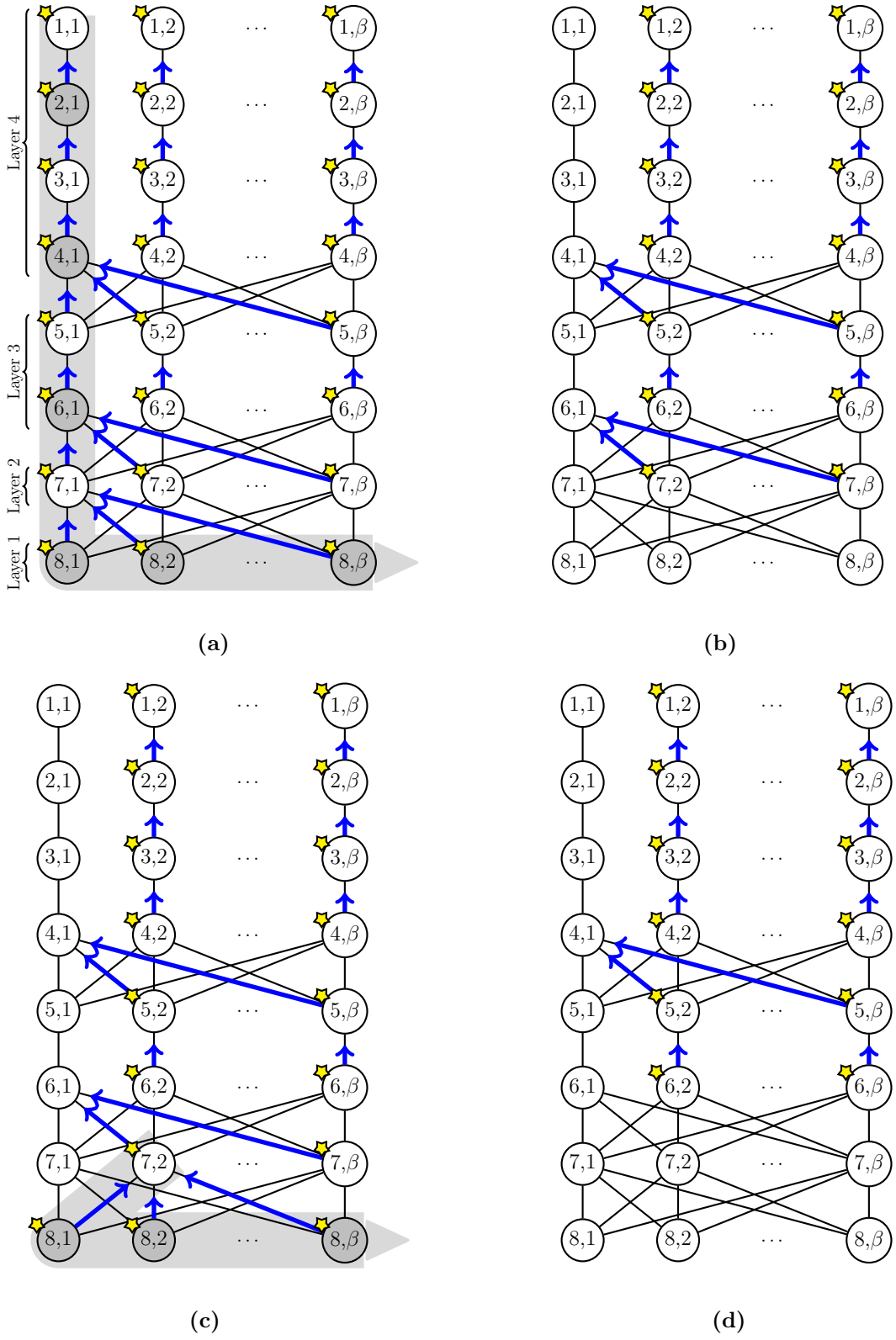
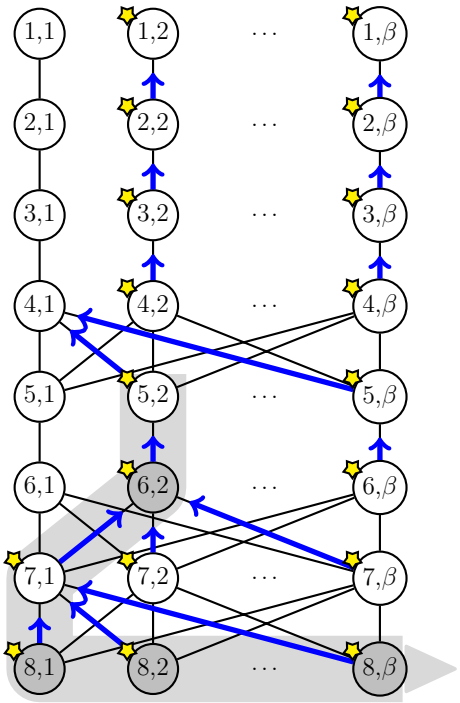
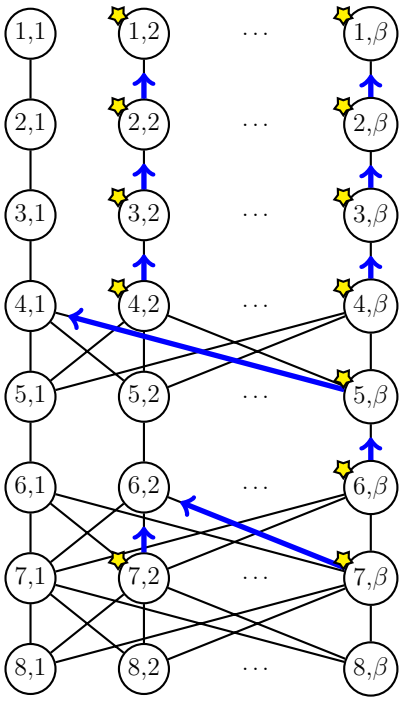


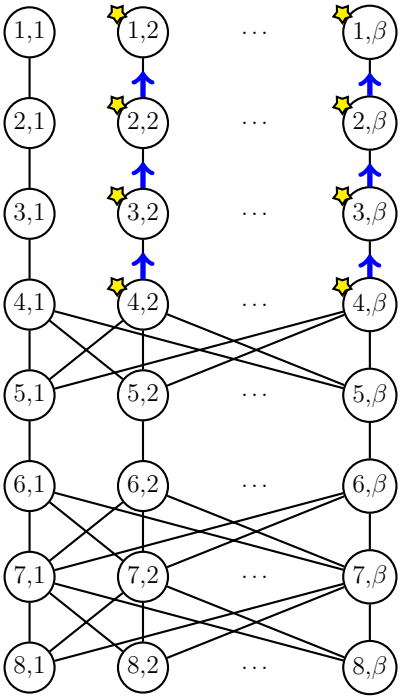
Figure 4.15 – Intuitive idea of the execution. The *leader* of a process is 0 if the process is labeled with a star, its own ID otherwise. *level* is not represented as it is always correct. The plain gray arrows show the processes that successively reset.



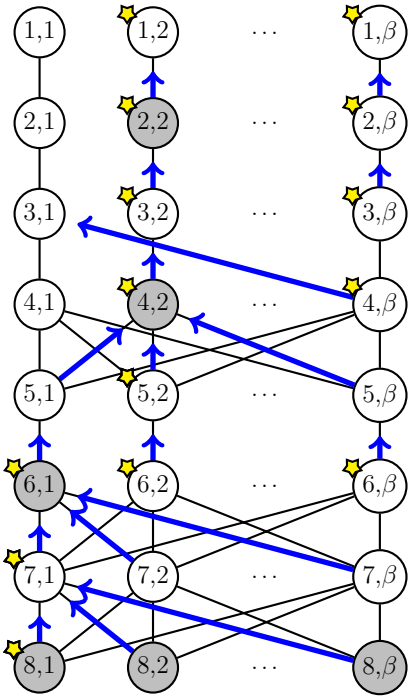
(e)



(f)



(g)



(h)

Figure 4.14 – (continued)

Algorithm 8 – Algorithm of the daemon.

```
1: function Daemon
2:     for i = 1...β, (i++) do
3:         Reset(3, i);
4:         if i < β then
5:             Build(3, i + 1);

6: function Reset(layer, col)
7:     for i = top(layer + 1)...bottom(layer + 1), (i++) do
8:         p_(i,col) executes R-action;
9:     if layer = 1 then
10:        for j = 1...β, (j++) do
11:            p_(L,j) executes R-action;                    ▷ Reset of layer 1
12:    else
13:        for j = 1...β, (j++) do
14:            Reset(layer − 1, j);
15:            if j < β then
16:                Build(layer − 1, j + 1);

17: function Build(layer, col)
18:    for i = top(layer)...bottom(layer), (i++) do
19:        for j = 1...β, (j++) do
20:            p_(i,j) executes J-action;
21:        for k = i − 1...2(i − L/2), (k−−) do
22:            if k ≥ top(layer) then
23:                for j = 1...β, (j++) do
24:                    p_(k,j) executes C1-action;
25:            else
26:                p_(k,col) executes C1-action;
27:        for k = i...2(i − L/2) + 1, (k−−) do
28:            if k ≥ top(layer) then
29:                for j = 1...β, (j++) do
30:                    p_(k,j) executes C2-action;
31:            else
32:                p_(k,col) executes C2-action;
33:    if layer > 1 then
34:        Build(layer − 1, 1);
```

4.5.3 Generalization to an Example of Execution in $\Omega(n^\alpha)$

We note E_4 the graph built for the example in $\Omega(n^4)$ steps and shown in Figure 4.15a. Then, starting from $E_{\alpha-1}$ ($\alpha \geq 5$), we can build E_α, a graph for which there exists an execution in $\Omega(n^\alpha)$ steps. The construction is based on the same principle as in Subsection 4.5.2, by adding a layer. If $E_{\alpha-1}$ has $L\beta$ processes $p_{(i,j)}$ ($1 \leq i \leq L, 1 \leq j \leq \beta$), then E_α has $L' = 2L$ lines of β processes $q_{(i',j')}$ ($1 \leq i' \leq L', 1 \leq j' \leq \beta$). The construction principle is as follows:

1. We increase the *level* and the ID of the $L\beta$ processes of $E_{\alpha-1}$ as follows: $\forall i \in \{1,\ldots,L\}, \forall j \in \{1,\ldots,\beta\}, q_{(i+L,j)} = p_{(i,j)}$. The ID of $q_{(i+L,j)}$ becomes $(i+L-1)\beta+j$ and $q_{(i+L,j)}.level = i + L$. The value of variables *color* and *done* do not change. If $i \neq 1$, the *par* remains the same. Otherwise, see step 3.

2. At the top of $E_{\alpha-1}$, we add L lines of β processes. These new processes satisfy:
   - $\forall i \in \{1,\ldots,L\}, \forall j \in \{1,\ldots,\beta\}$:
     - $q_{(i,j)}.id = (i-1)\beta + j$
     - $q_{(i,j)}.leader = 0$
     - $q_{(i,j)}.level = i$
     - $q_{(i,j)}.done = \text{FALSE}$
   - $\forall i \in \{2,\ldots,L\}, \forall j \in \{1,\ldots,\beta\}$, $\{q_{(i-1,j)}, q_{(i,j)}\} \in E$ and $q_{(i,j)}.par = q_{(i-1,j)}$
   - $\forall j \in \{1,\ldots,\beta\}, q_{(1,j)}.par = q_{(1,j)}$
   - $\forall j \in \{2,\ldots,\beta\}, \forall i \in \{1,\ldots,L\}$, $q_{(i,j)}.color = 2$
   - $\forall i \in \{1,\ldots,L\}$, $q_{(i,1)}.color = (i \mod 2) + 1$

3. The former first line of $E_{\alpha-1}$ becomes a new bipartite complete subgraph with the last added line:
   - $\forall j \in \{1,\ldots,\beta\}, \forall j' \in \{1,\ldots,\beta\}$, $\{q_{(L,j)}, q_{(L+1,j')}\} \in E$
   - $\forall j \in \{1,\ldots,\beta\}$, $q_{(L+1,j)}.par = q_{(L,1)}$

Figure 4.15 shows the structure of the network for E_5 and its initial configuration.

In the execution, the daemon selects processes according to function *Daemon*(α) (see Algorithm 9) which is the generalization of the algorithm presented in Section 4.5.2. In $E_{\alpha-1}$, processes $p_{(L,.)}$ execute $\beta^{\alpha-1}$ times **R**-action. Now, we added a new level of recursion. Then, processes $q_{(L',.)}$ execute β^α times **R**-action. As $\beta = \frac{n}{L'}$, the execution lasts $\Omega(n^\alpha)$ steps. Hence, we obtain:

Theorem 4.14

For every $\alpha \geq 4$, for every $\beta \geq 2$, there exists a network E_α of $n = 2^{\alpha-1} \times \beta$ processes in which there exists a possible execution of Algorithm $\mathcal{DLV}_2$ that stabilizes in $\Omega(n^\alpha)$ steps.

We proved that for E_α of size $n = L \times \beta$ ($\beta \geq 2$, $\alpha \geq 4$ and $L = 2^{\alpha-1}$), the execution in Algorithm 9 stabilizes using at least β^α steps. For a fixed size n of network, the value β^α

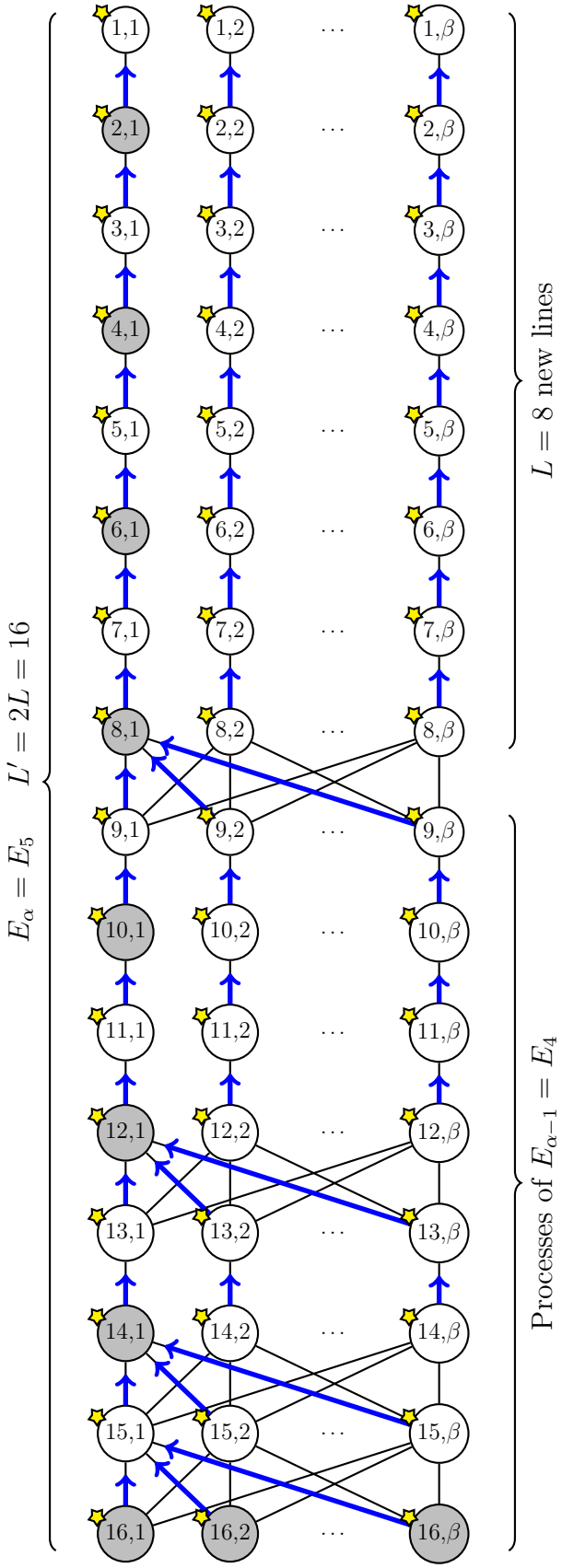


Figure 4.15 – Initial configuration of the example in $\Omega(n^5)$ steps.

Algorithm 9 – Generalization of the algorithm of the daemon for E_α.

```
1: function Daemon(α)
2:     for i = 1...β, (i++) do
3:         Reset(α − 1, i);                          ▷ See Algorithm 8
4:         if i < β then
5:             Build(α − 1, i + 1);                  ▷ See Algorithm 8
```

may vary, depending on *e.g.*, L. For instance, for $L = n/2$, we have that $\alpha = \log_2 n$ and $\beta = 2$ which implies that $\beta^\alpha = n$. At the opposite of the interval of L (second example), when $L = 8$, we have $\alpha = 4$ and $\beta = n \times 2^{-3}$. Hence, in this case, $\beta^\alpha = 2 \times n^4$.

Both costs obtained in those examples are polynomial. But, between them, the function reaches higher values: the following corollary shows that the highest value of β^α is reached for $L = \sqrt{\frac{n}{2}}$ and is non-polynomial.

Corollary 4.5

The stabilization time of Algorithm $\mathcal{DLV}_2$ is in $\Omega\left((2n)^{\frac{1}{4}\log_2(2n)}\right)$ steps.

Proof: We show below that for every $\alpha \geq 4$, for every $\beta \geq 2$, there exists a network of size $n = 2^{\alpha-1} \times \beta$ for which there exists an execution which stabilizes in $\Omega\left((2n)^{\frac{1}{4}\log_2(2n)}\right)$ steps.

Let β, α and L be positive integers such that $n = L \times \beta$, $\beta \geq 2$, $\alpha \geq 4$, and $L = 2^{\alpha-1}$ (as for Theorem 4.14). The value of the function β^α reaches its maximum when $L = \sqrt{\frac{n}{2}}$, $\beta = \sqrt{2n}$ and $\alpha = \frac{1}{2}(\log_2 n + 1)$. (This can be easily proved by cancellation of the derivative of β^α *w.r.t.* L.) In this case, β^α equals $(2n)^{\frac{1}{4}\log_2(2n)}$, and we are done. ■

4.6 Conclusion

Summary of Contributions. In this chapter, we have proposed a silent self-stabilizing leader election algorithm, called $\mathcal{LE}$, for static bidirectional connected identified networks of arbitrary topology. Starting from any arbitrary configuration, $\mathcal{LE}$ converges to a terminal configuration, where all processes know the ID of the leader, this latter being the process of minimum ID. Moreover, as in most of the solutions from the literature, a distributed spanning tree rooted at the leader is defined in the terminal configuration.

$\mathcal{LE}$ is written in the locally shared memory model. It assumes the distributed unfair daemon, the most general scheduling hypothesis of the model. Moreover, it requires no global knowledge on the network (such as an upper bound on the diameter or the number of processes, for example).

$\mathcal{LE}$ requires $\Theta(\log n + b)$ bits per process, where n is the size of the network and b is the number of bits requires to store an ID. If we consider that IDs are natural integers like it is commonly done in the literature, $b = \lceil \log n \rceil$. Hence, $\mathcal{LE}$ is asymptotically optimal in memory.

We have analyzed its stabilization time both in rounds and steps. We have shown that $\mathcal{LE}$ stabilizes in at most $3n + \mathcal{D}$ rounds, where $\mathcal{D}$ is the diameter of the network. We have also proven that for every $n \geq 4$, for every $\mathcal{D}, 2 \leq \mathcal{D} \leq n-2$, there is a network of n processes in which a possible execution exactly lasts this complexity.

Finally, we have proven that $\mathcal{LE}$ achieves a stabilization time polynomial in steps. More precisely, its stabilization time is at most $\frac{n^3}{2} + 2n^2 + \frac{n}{2} + 1$ steps. Then, we have shown for every $n \geq 4$, that there exists a network of n processes in which a possible execution exactly lasts $\frac{n^3}{6} + \frac{3}{2}n^2 - \frac{8}{3}n + 2$ steps, establishing then that the worst case is in $\Theta(n^3)$.

For fair comparison, we have studied the step complexity of the previous best algorithms with similar settings (*i.e.*, they do not use any global knowledge and are proven assuming an unfair daemon) given in [DLV11a, DLV11b] and respectively called here $\mathcal{DLV}_1$ and $\mathcal{DLV}_2$. We have shown that for any $n \geq 5$, there exists a network in which there is an execution of algorithm $\mathcal{DLV}_1$ that stabilizes in $\Omega(2^{\lfloor \frac{n-1}{4} \rfloor})$ steps. Hence, the stabilization time of $\mathcal{DLV}_1$ is not polynomial.

Similarly, we showed that for a given $\alpha \geq 3$, for every $\beta \geq 2$, there exists a network of $n = 2^\alpha \times \beta$ processes in which there is an execution of algorithm $\mathcal{DLV}_2$ that stabilizes in $\Omega(n^{\alpha+1})$. In other words, the stabilization time of $\mathcal{DLV}_2$ in steps is also not polynomial.

Perspectives. Perspectives of this work deal with complexity issues. In [DLV11b], Datta et al. showed that it is easy to implement a silent self-stabilizing leader election which works assuming an unfair daemon, uses $\Theta(\log n)$ bits per process, and stabilizes in $O(D)$ rounds (where D is an upper bound on $\mathcal{D}$). Nevertheless, processes are assumed to *know* D. It is worth investigating whether it is possible to design an algorithm which works assuming an unfair daemon, uses $\Theta(\log n)$ bits per process, and stabilizes in $O(\mathcal{D})$ rounds without using any global knowledge. We believe this problem remains difficult, even adding some fairness assumption.

Chapter 5

Gradual Stabilization under (τ, ρ)-dynamics and Unison

"Oh my ears and whiskers, how late it's getting!"
— Lewis Carroll, *Alice's Adventures in Wonderland*

Contents

5.1 Introduction

In this chapter, we propose and study a variant of self-stabilization designed to ensure fast convergence after topological changes in dynamic networks.

As stated before, *self-stabilization* [Dij74] is a general paradigm to enable the design

of distributed systems tolerating *any* finite number of transient faults. After the end of transient faults, a self-stabilizing system recovers *within finite time*, without any external intervention, a so-called legitimate configuration from which its specification is satisfied. It stabilizes in a unified manner, whatever the nature and extent of transient faults. Such versatility comes at a price, *e.g.*, that there is no safety guarantee during the stabilization phase. Hence, self-stabilizing algorithms are mainly compared according to their *stabilization time*, the maximum duration of the stabilization phase. For many problems, the stabilization time is significant, *e.g.*, for synchronization problems [AKM+93] and more generally for non-static problems [GT02] (such as token passing or broadcast) the lower bound is $\Omega(\mathcal{D})$ rounds, where $\mathcal{D}$ is the diameter of the network. By definition, the stabilization time is impacted by worst case scenarios. Now, in most cases, transient faults are sparse and their effect may be superficial. Recent research thus focuses on proposing self-stabilizing algorithms that additionally ensure drastically smaller convergence times in favorable cases.

Defining the number of faults hitting a network using some kind of Hamming distance (the minimal number of processes whose state must be changed in order to recover a legitimate configuration), variants of the self-stabilization paradigm have been defined, *e.g.*, a *time-adaptive* self-stabilizing algorithm [KP99] additionally guarantees a convergence time in $O(k)$ time-units when the initial configuration is at distance at most k from a legitimate configuration.

The property of *locality* consists in avoiding situations in which a small number of transient faults causes the entire system to be involved in a global convergence activity. Locality is, for example, captured by *fault containing* self-stabilizing algorithms [GGHP96], which ensure that when few faults hit the system, the faults are both spatially and temporally contained. "Spatially" means that if only few faults occur, those faults cannot be propagated further than a preset radius around the corrupted processes. "Temporally" means quick stabilization when few faults occur.

Some other approaches consist in providing convergence times *tailored by the type of transient faults*. For example, a *superstabilizing algorithm* [DH97] is self-stabilizing and has two additional properties when transient faults are limited to a single topological change. Indeed, after adding or removing one link or process in the network, a superstabilizing algorithm recovers fast (typically $O(1)$ rounds), and a safety predicate, called a *passage* predicate, should be satisfied all along the stabilization phase.

5.1.1 Contributions

We introduce a specialization of self-stabilization called *gradual stabilization under* (τ, ρ)*-dynamics* in Section 5.3. An algorithm is gradually stabilizing under (τ, ρ)-dynamics if it is self-stabilizing and satisfies the following additional feature. After up to τ *dynamic steps* of type ρ occur starting from a legitimate configuration, a gradually stabilizing algorithm first quickly recovers a configuration from which a specification offering a minimum quality of service is satisfied. It then gradually converges to specifications offering stronger and stronger safety guarantees until reaching a configuration:

- from which its initial (strong) specification is satisfied again, and

- where it is ready to achieve gradual convergence again in case of up to τ new dynamic steps of type ρ.

Of course, the gradual stabilization makes sense only if the convergence to every intermediate weaker specification is fast.

We illustrate this new property by considering three variants of a synchronization problem respectively called *strong*, *weak*, and *partial* unison. In these problems, each process should maintain a local clock. We restrict here our study to periodic clocks, *i.e.*, all local clocks are integer variables whose domain is $\{0, \ldots, \alpha - 1\}$, where $\alpha \geq 2$ is called the *period*. Each process should regularly increment its clock modulo α (liveness) while fulfilling some safety requirements. The safety of strong unison imposes that at most two consecutive clock values exist in any configuration of the system. Weak unison only requires that the difference between clocks of every two neighbors is at most one increment. Finally, we define partial unison as a property dedicated to dynamic systems, which only enforces the difference between clocks of neighboring processes present before the dynamic steps to remain at most one increment. The specifications of these problems are detailled in Section 5.2.2.

We propose in Section 5.5 a self-stabilizing strong unison algorithm $\mathcal{SU}$ which works with any period $\alpha \geq 2$ in an anonymous connected network of n processes. $\mathcal{SU}$ assumes the knowledge of two values μ and β, where μ is any value greater than or equal to $\max(2, n)$, α should divide β, and $\beta > \mu^2$. $\mathcal{SU}$ is designed in the locally shared memory model and assume the distributed unfair daemon, the most general daemon of the model. Its stabilization time is at most $n + (\mu + 1)\mathcal{D} + 1$ rounds, where n (resp. $\mathcal{D}$) is the size (resp. diameter) of the network.

We then slightly modify $\mathcal{SU}$ in Section 5.6 to make it gradually stabilizing under $(1, \mathsf{BULCC})$-dynamics. In particular, the parameter μ should now be greater than or equal to $\max(2, N)$, where N is a bound on the number of processes existing in any reachable configuration. Our gradually stabilizing variant of $\mathcal{SU}$ is called $\mathcal{DSU}$. Due to the slight modifications, the stabilization time of $\mathcal{DSU}$ is increased by one round compared to the one of $\mathcal{SU}$. The condition BULCC restricts the gradual convergence obligation to dynamic steps, called BULCC-dynamic steps, that fulfill the following conditions. A BULCC-dynamic step may contain several topological events, *i.e.*, link and/or process additions and/or removals. However, after such a step, the network should:

1. contains at most N processes,
2. stay connected, and
3. if $\alpha > 3$, every process which joins the system should be linked to at least one process already in the system before the dynamic step, unless all of those have left the system.

Condition 1) is necessary to have finite periodic clocks in $\mathcal{DSU}$. In Section 5.4, we show the necessity of condition 2) to obtain our results whatever the period is, while we proved that condition 3) is necessary for our purposes when the period α is fixed to a value

greater than 5. Finally, we exhibit pathological cases for periods 4 and 5, in case we do not assume condition 3).

$\mathcal{DSU}$ is gradually stabilizing because after one BULCC-dynamic step from a configuration which is legitimate for the strong unison, it immediately satisfies the specification of partial unison, then converges to the specification of weak unison in at most one round, and finally retrieves, after at most $(\mu + 1)\mathcal{D}_1 + 1$ additional rounds (where $\mathcal{D}_1$ is the diameter of the network after the dynamic step), a configuration:

- from which the specification of strong unison is satisfied, and
- where it is ready to achieve gradual convergence again in case of another dynamic step.

Notice that $\mathcal{DSU}$ being also self-stabilizing (by definition), it still converges to a legitimate configuration of the strong unison after the system suffers from arbitrary other kinds of transient fault including, for example, several arbitrary dynamic steps. However, in such cases, there is no safety guarantees during the stabilization phase.

A preliminary version of these contributions presenting conditions on the dynamic steps and the algorithm for $\alpha \geq 4$ is published in the proceedings of the 22nd International Conference on Parallel and Distributed Computing (Euro-Par 2016) [ADDP16].

5.1.2 Related Work

Gradual stabilization is related to two other stronger forms of self-stabilization, namely, *safe-converging self-stabilization* [KM06] and *superstabilization* [DH97]. The goal of a safely converging self-stabilizing algorithm is to first quickly (within $O(1)$ rounds is the usual rule) converge from an arbitrary configuration to a *feasible* legitimate configuration, where a minimum quality of service is guaranteed. Once such a feasible legitimate configuration is reached, the system continues to converge to an *optimal* legitimate configuration, where more stringent conditions are required. Hence, the aim of safe-converging self-stabilization is also to ensure a gradual convergence, but only for two specifications. However, such a gradual convergence is stronger than ours as it should be ensured after any step of transient faults,[1] while the gradual convergence of our property applies after dynamic steps only. Safe convergence is especially interesting for self-stabilizing algorithms that compute optimized data structures, *e.g.*, minimal dominating sets [KM06], approximately minimum weakly connected dominating sets [KK08], approximately minimum connected dominating sets [KIY13, KK12], and minimal (f, g)-alliances [CDD+15]. However, to the best of our knowledge, no safe-converging algorithm for non-static problems, such as unison for example, has been proposed until now.

In superstabilization, like in our approach, fast convergence and the passage predicate should be ensured only if the system was in a legitimate configuration before the topological change occurs. In contrast with our approach, superstabilization ensures fast convergence to the original specification. However, this strong property only considers one dynamic step consisting in only one topological event: the addition or removal of one link or process in the network. Again, superstabilization has been especially studied in the

[1] Such transient faults may include topological changes, but not only.

context of static problems, *e.g.*, spanning tree construction [DH97, BPRT10, BPR13], and coloring [DH97]. However, notice that there exist few superstabilizing algorithms for non-static problems in particular topologies, *e.g.*, mutual exclusion in rings [Her00, KUFM02].

We use the general term *unison* to name several close problems also known in the literature as *phase or barrier synchronization* problems. There exist many self-stabilizing algorithms for the strong as well as weak unison problem, *e.g.*, [Bou07, GH90, ADG91, HL98, NV01, JADT02, BPV04, TJH10]. However, to the best of our knowledge, until now, no self-stabilizing solution for such problems addresses specific convergence properties in case of topological changes (in particular, no superstabilizing ones). Self-stabilizing strong unison was first considered in synchronous anonymous networks. Particular topologies were considered in [HL98] (rings) and [NV01] (trees). Gouda and Herman [GH90] proposed a self-stabilizing algorithm for strong unison working in anonymous synchronous systems of arbitrary connected topology. However, they considered unbounded clocks. A solution working with the same settings, yet implementing bounded clocks, is proposed in [ADG91]. In [TJH10], an asynchronous self-stabilizing strong unison algorithm is proposed for arbitrary connected rooted networks.

Johnen et al. investigated asynchronous self-stabilizing weak unison in oriented trees in [JADT02]. The first self-stabilizing asynchronous weak unison for general graphs was proposed by Couvreur et al. [CFG92]. However, no complexity analysis was given. Another solution which stabilizes in $O(n)$ rounds has been proposed by Boulinier et al. in [BPV04]. Finally, Boulinier proposed in his PhD thesis a parametric solution which generalizes both the solutions of [CFG92] and [BPV04]. In particular, the complexity analysis of this latter algorithm reveals an upper bound in $O(\mathcal{D}.n)$ rounds on the stabilization time of the Couvreur et al.' algorithm.

5.2 Preliminaries

In this section, we detail the considered context (Section 5.2.1) and we define the specifications of the considered synchronization problems (Section 5.2.2).

5.2.1 Context

We consider dynamic and bidirectional networks of anonymous processes. We assume that $\mathcal{G}_0$, the initial topology of the system, is arbitrary yet connected, contains $n \geq 1$ processes, and its diameter is $\mathcal{D}$. We consider the locally shared memory model presented in Section 2.6 and the distributed unfair daemon.

5.2.2 Unison

We consider three close synchronization problems included here under the general term of *unison*. In these problems, each process should maintain a local *clock*. We restrict here our study to periodic clocks: α, called the *period* of the clocks, should be greater than or equal to 2. The aim is to make all local clocks regularly incrementing (modulo α) in a finite set of integer values $\{0, \ldots, \alpha - 1\}$ while fulfilling some safety requirements.

All these problems require the same liveness property which means that whenever a clock has a value in $\{0, \ldots, \alpha - 1\}$, it should eventually increment.

Definition 5.1 (*Liveness of the Unison*)

An execution $e = (\gamma_i)_{i \geq 0}$ satisfies the liveness property LIVE if and only if:
$\forall \gamma_i \in e, \forall p \in V_i, \forall x \in \{0, \ldots, \alpha - 1\}$,

$$\gamma_i(p).clock = x \;\Rightarrow\; \exists j > i, (\forall k \in \{i+1, \ldots, j-1\}, p \in V_k \wedge \gamma_k(p).clock = x) \wedge (p \in V_j \wedge \gamma_j(p).clock = (x+1) \bmod \alpha)$$

The three versions of unison we consider are respectively named *strong*, *weak*, and *partial* unison, and differ by their safety property. *Strong unison* is also known as the *phase* or *barrier* synchronization problem [Mis91, KA97]. The *weak unison* appeared first in [CFG92] under the name of *asynchronous unison*. We define the *partial unison* as a straightforward variant of the weak unison suited for dynamic systems.

Definition 5.2 (*Safety of the Partial Unison*)

An execution $e = (\gamma_i)_{i \geq 0}$ satisfies the safety property SAFE_{PU} if and only if $\forall \gamma_i \in e$, the following conditions holds

- $\forall p \in V_i \setminus New(i), \gamma_i(p).clock \in \{0, \ldots, \alpha - 1\}$ and
- $\forall p \in V_i \setminus New(i), \forall q \in \gamma_i(p).\mathcal{N} \setminus New(i)$,

$$\gamma_i(p).clock \in \{\gamma_i(q).clock, (\gamma_i(q).clock + 1) \bmod \alpha, (\gamma_i(q).clock - 1) \bmod \alpha\}$$

meaning that the clocks of any two neighbors which are not in bootstate[2] differ from at most one increment (modulo α).

Definition 5.3 (*Safety of the Weak Unison*)

An execution $e = (\gamma_i)_{i \geq 0}$ satisfies the safety property SAFE_{WU} if and only if

- $\forall \gamma_i \in e$, $New(i) = \emptyset$, meaning that no process is in bootstate and
- $\text{SAFE}_{\text{PU}}(e)$ holds.

In the next definition, we use the following notation: for every configuration γ_i, let $CV(\gamma_i) = \{\gamma_i(p).clock : p \in V_i\}$ be the set of clock values present in configuration γ_i.

[2]Recall that while a process is in bootstate, it has not taken any step and so its output, here its clock value, is meaningless.

Definition 5.4 (*Safety of the Strong Unison*)

An execution $e = (\gamma_i)_{i \geq 0}$ satisfies the safety property SAFE_{SU} if and only if $\forall \gamma_i \in e$, the following conditions holds

- $New(i) = \emptyset$, meaning that no process is in bootstate,
- $\forall p \in V_i, \gamma_i(p).clock \in \{0, \ldots, \alpha - 1\}$, and
- $|CV(\gamma_i)| \leq 2 \wedge (CV(\gamma_i) = \{x, y\} \Rightarrow x = (y + 1) \bmod \alpha \vee y = (x + 1) \bmod \alpha)$, meaning that there exists at most two different clock values, and if so, these two values are consecutive (modulo α).

Then, using Definitions 5.1-5.4, we define the specifications of partial unison, weak unison, and strong unison, denoted SP_{PU}, SP_{WU}, and SP_{SU}, respectively.

Definition 5.5 (*Partial Unison*)

An execution e of algorithm ALG satisfies the specification SP_{PU} of the *partial unison* problem if and only if $\text{LIVE}(e) \wedge \text{SAFE}_{\text{PU}}(e)$ holds.

Definition 5.6 (*Weak Unison*)

An execution e of algorithm ALG satisfies the specification SP_{WU} of the *weak unison* problem if and only if $\text{LIVE}(e) \wedge \text{SAFE}_{\text{WU}}(e)$ holds.

Definition 5.7 (*Strong Unison*)

An execution e of algorithm ALG satisfies the specification SP_{SU} of the *strong unison* problem if and only if $\text{LIVE}(e) \wedge \text{SAFE}_{\text{SU}}(e)$ holds.

The property below sum up the straightforward relationship between the three variants of unison we consider here.

Property 5.1

For every execution e, we have $SP_{\text{SU}}(e) \Rightarrow SP_{\text{WU}}(e) \Rightarrow SP_{\text{PU}}(e)$.

5.3 Gradual Stabilization under (τ, ρ)-dynamics

Below, we introduce a specialization of self-stabilization called *gradual stabilization under* (τ, ρ)*-dynamics*. The overall idea behind this concept is to design self-stabilizing algorithms that ensure additional properties (stronger than "simple" eventual convergence) when the system suffers from topological changes. Initially observe the system from a legitimate configuration, and assume that up to τ ρ-dynamic steps occur. The very first configuration after those steps may be illegitimate, but this configuration is usually far from being arbitrary. In that situation, the goal of gradual stabilization is to first quickly recover a configuration from which a weaker specification offering a minimum quality of service is satisfied and then make the system gradually re-stabilizes to stronger and stronger specifications, until fully recovering its initial (strong) specification. Of course, the gradual stabilization makes sense only if the convergence to every intermediate weaker specification is fast and each of those weak specifications offers a useful interest.

5.3.1 Definition

Let $\tau \geq 0$. For every execution $e = (\gamma_i)_{i \geq 0} \in \mathcal{E}^\tau$ (*i.e.*, e contains at most τ dynamic steps), we note $\gamma_{fst(e)}$ the first configuration of e after the last dynamic step. Formally, $fst(e) = \min\{i : (\gamma_j)_{j \geq i} \in \mathcal{E}^0\}$. For any subset E of $\mathcal{E}^\tau$, let $FC(E) = \left\{\gamma_{fst(e)} : e \in E\right\}$ be the set of all configurations that can be reached after the last dynamic step in executions of E.

Let $SP_1, SP_2, \ldots, SP_k$, be an ordered sequence of specifications. Let $B_1, B_2, \ldots, B_k$ be (asymptotic) complexity bounds such that $B_1 \leq B_2 \leq \cdots \leq B_k$. Let ρ be a dynamic pattern.

Definition 5.8 (*Gradual Stabilization under (τ, ρ)-dynamics*)

A distributed algorithm Alg is *gradually stabilizing under* (τ, ρ)*-dynamics* for $(SP_1 \bullet B_1, SP_2 \bullet B_2, \ldots, SP_k \bullet B_k)$ if and only if $\exists \mathscr{L}_1, \ldots, \mathscr{L}_k \subseteq \mathcal{C}$ such that

1. Alg stabilizes from $\mathcal{C}$ to SP_k by $\mathscr{L}_k$.
2. $\forall i \in \{1, \ldots, k\}$,
   - Alg stabilizes from $FC(\mathcal{E}^{\tau,\rho}_{\text{Alg}}(\mathscr{L}_k))$ to SP_i by $\mathscr{L}_i$, and
   - the convergence time in rounds from $FC(\mathcal{E}^{\tau,\rho}_{\text{Alg}}(\mathscr{L}_k))$ to $\mathscr{L}_i$ is bounded by B_i.

The first point ensures that a gradually stabilizing algorithm is still self-stabilizing for its strongest specification. Hence, its performances can be also evaluated at the light of its stabilization time. Indeed, it captures the maximal convergence time of the gradually stabilizing algorithm after the system suffers from an arbitrary finite number of transient faults (those faults may include an unbounded number of arbitrary dynamic steps, for example).

The second point means that after up to τ ρ-dynamic steps from a configuration that is legitimate w.r.t. the strongest specification SP_k, the algorithm *gradually converges* to each specification SP_i with $i \in \{1, \ldots, k\}$ in at most B_i rounds.

Note that B_k captures a complexity similar to the *fault gap* in fault-containing algorithms [GGHP96]: assume a period P_1 of up to τ ρ-dynamic steps starting from a legitimate configuration of $\mathscr{L}_k$; B_k represents the necessary fault-free interval after P_1 and before the next period P_2 of at most τ ρ-dynamic steps so that the system converges to a legitimate configuration of $\mathscr{L}_k$ and so becomes ready again to achieve gradual convergence after P_2.

5.3.2 Related Properties

Gradual stabilization is related to two other stronger forms of self-stabilization: *safe-converging self-stabilization* [KM06] and *superstabilization* [DH97].

As stated in the related work (Section 5.1.2), the aim of a safely converging self-stabilizing algorithm is to ensure a gradual convergence, but for only two specifications.

However, this kind of gradual convergence should be ensure after any step of transient faults (such transient faults can include topological changes, but not only), while the gradual convergence of our property applies after dynamic steps only.

Like in our approach, a superstabilizing algorithm ensures fast convergence after a dynamic step, if the system was in a legitimitate configuration before the topological changes (specification recovered in $O(1)$ round, passage predicate during the convergence). In contrast with our approach, superstabilization consists in only *one* dynamic step satisfying a very restrictive dynamic pattern, noted here ρ_1: only one topological event, *i.e.*, the addition or removal of one link or process in the network. A superstabilizing algorithm for a specification SP_1 can be seen as an algorithm which is gradually stabilizing under $(1, \rho_1)$-dynamics for $(SP_0 \bullet 0, SP_1 \bullet f)$ where SP_0 is the passage predicate and f is the superstabilization time.

5.4 Conditions on the Dynamic Pattern

In Section 5.6, we provide a gradually stabilizing algorithm under $(1, \mathsf{BULCC})$-dynamics for $(SP_{\text{PU}} \bullet 0, SP_{\text{WU}} \bullet 1, SP_{\text{SU}} \bullet B)$, denoted $\mathcal{DSU}$, where B is a given complexity bound, starting from any arbitrary anonymous (initially connected) network and assuming the distributed unfair daemon. The dynamic pattern BULCC (see Definition 5.6 on page 155) requires, in particular, that graphs remain *Connected* (*i.e.*, the dynamic pattern C below) and, if the period α of unison is greater than 3, then the condition *Under Local Control* (*i.e.*, the dynamic pattern ULC below) should hold:

- $\mathsf{C}(G_i, G_j) \equiv$ if graph G_i is connected, then graph G_j is connected.

  Notice that as the initial topology of the system is assumed to be connected, the topology is always connected along any execution of $\mathcal{E}^{1,\mathsf{C}}$.

- $\mathsf{ULC}(G_i, G_j) \equiv$ if $V_i \cap V_j \neq \emptyset$ and G_i is connected, then $V_i \cap V_j$ is a dominating set of G_j.

  A *dominating set* of the graph $G = (V, E)$ is any subset D of V such that every node not in D is adjacent to at least one member of D.

  ULC permits to prevent a notable desynchronization of clocks. Namely, if not all processes leave the system during a dynamic step $\gamma \mapsto^d \gamma'$ from an initially connected topology, then every process that joins the system during that dynamic step is required to be "under the control of" (that is, linked to) at least one process which exists in both γ and γ'.

We now study the necessity of conditions C and ULC. We first show that the assumption C on dynamic steps is necessary whatever the value of the period α is (Theorem 5.1). We then show that the dynamic pattern ULC is necessary for any period $\alpha > 5$ (Theorem 5.2), while our algorithm shows that ULC is not necessary for each period $\alpha < 4$. For remaining cases (periods 4 and 5), our answer is partial, as we show that there are pathological cases among possible dynamic steps which satisfy C but not ULC (Theorem 5.4 and Corollary 5.1) that make any algorithm fails to solve our problem. In particular, for the case $\alpha = 5$, we exhibit an important class of such pathological dynamic steps (Theorem 5.3).

General Proof Context. To prove the above results, we assume from now on the existence of a deterministic algorithm ALG which is gradually stabilizing under (1,ρ)-dynamics for $(SP_{\text{PU}} \bullet 0, SP_{\text{WU}} \bullet 1, SP_{\text{SU}} \bullet B)$ starting from any arbitrary anonymous (initially connected) network under the distributed unfair daemon, where ρ is a given dynamic pattern and B is any (asymptotic) strictly positive complexity bound. Hence, our proofs consist in showing properties that ρ should satisfy (w.r.t. dynamic patterns C and ULC) in order to prevent Algorithm ALG from failing. In the sequel, we also denote by $\mathscr{L}_{\text{SU}}$ the legitimate configurations of ALG w.r.t. specification SP_{SU}.

5.4.1 Connectivity

Theorem 5.1

For every graph $\mathcal{G}$ and $\mathcal{G}'$, we have $\rho(\mathcal{G}, \mathcal{G}') \Rightarrow \mathsf{C}(\mathcal{G}, \mathcal{G}')$.

Proof: By contradiction, assume that there exists two graphs $\mathcal{G}_i$ and $\mathcal{G}_j$ such that $\rho(\mathcal{G}_i, \mathcal{G}_j)$, $\mathcal{G}_i$ is connected, and $\mathcal{G}_j$ is disconnected. Then, there is an execution $e = (\gamma_i)_{i \geq 0} \in \mathcal{E}^{1,\rho}_{\text{ALG}}(\mathscr{L}_{\text{SU}})$ such that $\mathcal{G}_0 = \mathcal{G}_i$ and $\mathcal{G}_{fst(e)} = \mathcal{G}_j$. Let A and B be two connected components of $\mathcal{G}_{fst(e)}$. By definition, there exists $j \geq fst(e)$ such that $\gamma_j \in \mathscr{L}_{\text{SU}}$ and A and B are defined in all configurations $(\gamma_i)_{i \geq j}$. From γ_j, all processes regularly increment their clocks in both A and B by the liveness property of strong unison. Now, as no process of B is linked to any process of A, the behavior of processes in B has no impact on processes in A and vice versa. So, liveness implies, in particular, that there always exists enabled processes in A. Consequently, there exists a possible execution of $\mathcal{E}^{1,\rho}_{\text{ALG}}(\mathscr{L}_{\text{SU}})$ prefixed by $\gamma_0 \dots \gamma_j$ where the distributed unfair daemon only selects processes in A from γ_j, hence violating the liveness property of strong unison, a contradiction. ■

5.4.2 Under Local Control

Technical Results. The following property states that, whenever $\alpha > 3$, once a legitimate configuration of the strong unison is reached, the system necessarily goes through a configuration where all clocks have the same value between any two increments at the same process.

Property 5.2

Assume $\alpha > 3$. For every $(\gamma_i)_{i \geq 0} \in \mathcal{E}^0_{\text{ALG}}(\mathscr{L}_{\text{SU}})$, for every process p, for every $k \in \{0, \dots, \alpha - 1\}$, for every $i \geq 0$, if p increments its clock from k to $(k+1) \mod \alpha$ in $\gamma_i \mapsto^s \gamma_{i+1}$ and $\exists j > i+1$ such that $\gamma_j(p).clock = (k+2) \mod \alpha$, then there exists $x \in \{i+1, \dots, j-1\}$, such that all clocks have value $(k+1) \mod \alpha$ in γ_x.

Proof: Let $(\gamma_i)_{i \geq 0} \in \mathcal{E}^0_{\text{ALG}}(\mathscr{L}_{\text{SU}})$ and p be a process. Let $k \in \{0, \dots, \alpha - 1\}$ and $i \geq 0$ such that p increments its clock from k to $(k+1) \mod \alpha$ in $\gamma_i \mapsto^s \gamma_{i+1}$ and $\exists j > i+1$ such that $\gamma_j(p).clock = (k+2) \mod \alpha$.

Assume, by the contradiction, that there is a process q such that $\gamma_i(q).clock = (k-1) \mod \alpha$. As the daemon is distributed and unfair, there is a possible static step where p moves, but not q leading to a configuration where $q.clock = (k-1) \mod \alpha$ and

$p.clock = (k+1) \mod \alpha$. This configuration violates the safety of SP_{SU}. Hence, there exists an execution of $\mathcal{E}^0_{\text{ALG}}(\mathscr{L}_{\text{SU}})$ which does not satisfy SP_{SU}, a contradiction.

Hence, $\forall q \in V, \gamma_i(q).clock \in \{k, (k+1) \mod \alpha\}$, by the safety of SP_{SU}. Similarly to the previous case, while there are processes whose clock value is k, no process (in particular p) can increment its clock from $(k+1) \mod \alpha$ to $(k+2) \mod \alpha$. Hence, between γ_{i+1} (included) and γ_{j-1} (included), there exists a configuration where all processes have clock value $(k+1) \mod \alpha$, since $\gamma_j(p).clock = (k+2) \mod \alpha$. ■

Since Alg is gradually stabilizing under (1,ρ)-dynamics for $(SP_{\text{PU}} \bullet 0, SP_{\text{WU}} \bullet 1, SP_{\text{SU}} \bullet B)$, follows.

Remark 5.1

Every execution in $\mathcal{E}^{1,\rho}_{\text{ALG}}$ is infinite.

Lemma 5.1

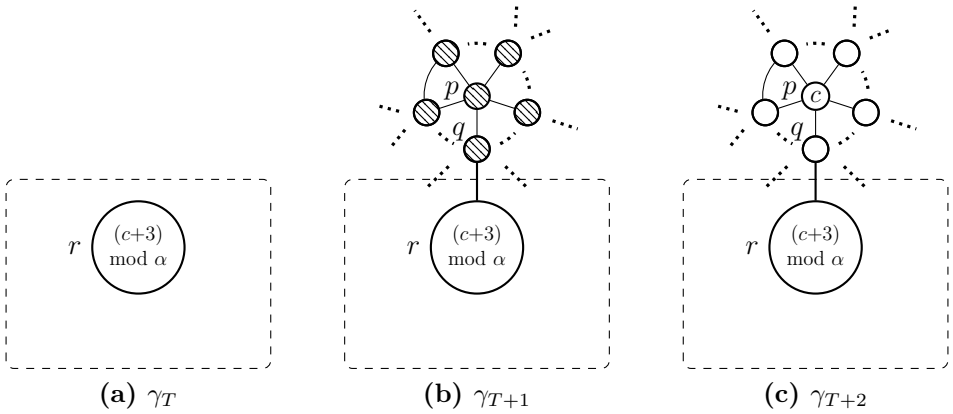
Let $\gamma_i \mapsto^{d,\rho} \gamma_{i+1}$ be a ρ-dynamic step such that $\gamma_i \in \mathscr{L}_{\text{SU}}$ and $\mathcal{G}_i$ is connected. For every process $p \in New(i+1)$, p is enabled in γ_{i+1} and if p moves, then in the next configuration, p is not bootstate and $p.clock \in \{0, \ldots, \alpha-1\}$.

Proof: As $\gamma_i \in \mathscr{L}_{\text{SU}}$ and $\mathcal{G}_i$ is connected, there is an execution $\mathcal{E}^{1,\rho}_{\text{ALG}}(\mathscr{L}_{\text{SU}})$ prefixed by $\gamma_i\gamma_{i+1}$. Moreover, there are enabled processes in γ_{i+1}, by Remark 5.1 and the fact that no more dynamic step occurs from γ_{i+1}. Assume that the daemon makes a synchronous static step from γ_{i+1}. The step $\gamma_{i+1} \mapsto^s \gamma_{i+2}$ actually corresponds to a complete round, by definition. So, the execution suffix from γ_{i+2} should satisfy the specification of the weak unison (any execution of $\mathcal{E}^{1,\rho}_{\text{ALG}}(\mathscr{L}_{\text{SU}})$ prefixed by $\gamma_i\gamma_{i+1}$ should converge in one round from $\gamma_{fst(e)} = \gamma_{i+1}$ to a configuration that is legitimate w.r.t. SP_{WU}). Now, if, by the contradiction, p is disabled in γ_{i+1}, or p is not bootstate in γ_{i+2}, or $\gamma_{i+2}(p).clock \notin \{0, \ldots, \alpha-1\}$, then the safety of the weak unison is violated in γ_{i+2}, a contradiction. ■

Lemma 5.2

Let $c \in \{0, \ldots, \alpha-1\}$, $\mathcal{G}$ be a connected graph of at least two nodes, and r_1 and r_2 be two nodes of $\mathcal{G}$. If $\alpha > 3$, then there exists an execution $e \in \mathcal{E}^0_{\text{ALG}}(\mathscr{L}_{\text{SU}})$ on the graph $\mathcal{G}$ which contains a configuration γ_T where r_1 and r_2 have two different clock values, one being $c \mod \alpha$ and the other $(c+1) \mod \alpha$.

Proof: Consider an execution e' in $\mathcal{E}^0_{\text{ALG}}(\mathscr{L}_{\text{SU}})$ on the graph $\mathcal{G}$. The specification of the strong unison is satisfied in e' and by liveness and Property 5.2, there is a configuration γ_S in e' where every clock equals $c \mod \alpha$. By liveness again, from γ_S, eventually there is a step where either r_1, or r_2, or both increments to $(c+1) \mod \alpha$. Consider the first step $\gamma_{z-1} \mapsto^s \gamma_z$ after γ_S, where either r_1, or r_2, or both increments to $(c+1) \mod \alpha$. In the two first cases, let $\gamma_T = \gamma_z$ and $e = e'$. For the last case, consider an execution e'' of $\mathcal{E}^0_{\text{ALG}}(\mathscr{L}_{\text{SU}})$ with the prefix $\gamma_0 \ldots \gamma_{z-1}$ common to e', but only r_1 moves in the step from γ_{z-1}. Let γ_T be the configuration reached by this latter step and $e = e''$. In either cases, r_1 and r_2 have two different clock values in γ_T, one being $c \mod \alpha$ and the other $(c+1) \mod \alpha$. ■



(a) γ_T **(b)** γ_{T+1} **(c)** γ_{T+2}

Figure 5.1 – Execution e'' in the proof of Theorem 5.2. The hachured nodes are in bootstate. The value inside the node is the value of its clock. If there is no value, its clock value is meaningless.

Lemma 5.3

Let $\mathcal{G}$ be any connected graph. There exists $\gamma_i \in \mathscr{L}_{\text{SU}}$ such that $\mathcal{G}_i = \mathcal{G}$.

Proof: ALG being designed for arbitrary initially connected networks, there exists at least one execution $e = (\gamma_i)_{i \geq 0} \in \mathcal{E}^0_{\text{ALG}}$, where $\mathcal{G}_i = \mathcal{G}$, $\forall i \geq 0$. By hypothesis, at least one configuration of e belongs to $\mathscr{L}_{\text{SU}}$. ■

Main Results.

Theorem 5.2

If $\alpha > 5$, then for every graphs $\mathcal{G}$ and $\mathcal{G}'$, we have $\rho(\mathcal{G}, \mathcal{G}') \Rightarrow \mathsf{C}(\mathcal{G}, \mathcal{G}') \wedge \mathsf{ULC}(\mathcal{G}, \mathcal{G}')$.

Proof: We illustrate the following proof with Figure 5.1. Assume $\alpha > 5$ and let $\mathcal{G}_{x-1}$ and $\mathcal{G}_x$ be two graphs such that $\rho(\mathcal{G}_{x-1}, \mathcal{G}_x)$. By Theorem 5.1, $\mathsf{C}(\mathcal{G}_{x-1}, \mathcal{G}_x)$ holds. So, assume, by the contradiction, that $\neg\mathsf{ULC}(\mathcal{G}_{x-1}, \mathcal{G}_x)$. Then, $\mathsf{C}(\mathcal{G}_{x-1}, \mathcal{G}_x) \wedge \neg\mathsf{ULC}(\mathcal{G}_{x-1}, \mathcal{G}_x)$ implies, in particular, that both $\mathcal{G}_{x-1}$ and $\mathcal{G}_x$ are connected.

By Lemma 5.3, there exists a configuration $\gamma_{x-1} \in \mathscr{L}_{\text{SU}}$, whose topology is $\mathcal{G}_{x-1}$. Consider now the configuration γ_x of topology $\mathcal{G}_x$, such that $\gamma_{x-1} \mapsto^{d,\rho} \gamma_x$ is a ρ-dynamic step that contains no process activation. Then, since $\mathcal{G}_x$ is connected and $V_{x-1} \cap V_x \neq \emptyset$ is not a dominating set, we have: $\exists p \in V_x \setminus V_{x-1}$ such that

1. $\forall v \in \gamma_x(p).\mathcal{N}$, $v \in V_x \setminus V_{x-1}$ and
2. there is a process $q \in \gamma_x(p).\mathcal{N}$ which has at least one neighbor in $V_{x-1} \cap V_x$, say r.

Moreover, p and its neighbors (in particular q) are in bootstate in γ_x. So, by Lemma 5.1, they all are enabled in γ_x and if they move, they will be no more in bootstate and their clock value will belong to $\{0, \ldots, 4\}$ in the configuration that follows γ_x. Let c be the clock value of p in the next configuration, if p moves.

By the liveness property of the strong unison, there exists an execution e in $\mathcal{E}^0(\mathscr{L}_{\text{SU}})$ on the graph $\mathcal{G}_{x-1}$ (*n.b.*, $\mathcal{G}_{x-1}$ is connected) which contains a configuration γ_T where r has clock value $(c+3) \mod \alpha$, see Figure 5.1a.

Consider now another execution $e' \in \mathcal{E}^{1,\rho}_{\text{ALG}}(\mathscr{L}_{\text{SU}})$ having a prefix common to e until γ_T. Assume that the unfair daemon introduces a ρ-dynamic step (it is possible since there was no dynamic step until now). Since $\mathcal{G}_T = \mathcal{G}_{x-1}$, the daemon can choose a step $\gamma_T \mapsto^{d,\rho} \gamma_{T+1}$, where no process moves and $\mathcal{G}_{T+1} = \mathcal{G}_x$. Now, $\forall v \in V_{T+1} \setminus V_T$, $\gamma_{T+1}(v) = \gamma_x(v)$, so again, in γ_{T+1}, p and all its neighbors (in particular q) are in bootstate and enabled. Moreover, if they move, they will be not in bootstate and their clock value will belong to $\{0, \ldots, 4\}$ in γ_{T+2}, by Lemma 5.1. Moreover, p is in the same situation as in γ_x, so if it moves, its clock is equal to c in γ_{T+2}. Then, r is still a neighbor of q which is still not in bootstate and still with clock value $(c+3) \mod 5$, see Figure 5.1b. By definition, since strong unison is satisfied in γ_T (by assumption), the partial unison necessarily holds all along the suffix of e' starting at γ_{T+1}. Assume that the daemon exactly selects p and its neighbors in the next static step $\gamma_{T+1} \mapsto^s \gamma_{T+2}$. In γ_{T+2} (Figure 5.1c), r is still not in bootstate and its clock is still equal to $(c+3) \mod 5$, since it did not move. Moreover, p is no more in bootstate and its clock equals c. Now, in γ_{T+2}, q is no more in bootstate and its clock value belongs to $\{0, \ldots, 4\}$. That clock value should differ of at most one increment (mod 5) from the clocks of p and r since partial unison holds in γ_{T+1} and all subsequent configurations. If the clock of q equals:

- c or $(c+1) \mod \alpha$, the difference between the clocks of q and r is at least 2 increments (mod α),
- $(c+2) \mod \alpha$, $(c+3) \mod \alpha$, $(c+4) \mod \alpha$, the difference between the clocks of q and p is at least 2 increments (mod α),
- any value in $\{0, \ldots, \alpha-1\} \setminus \{c, (c+1) \mod \alpha, (c+2) \mod \alpha, (c+3) \mod \alpha, (c+4) \mod \alpha\}$, the difference between the clocks of q and r is at least 2 increments (mod α).

Hence, the safety of partial unison is necessarily violated in the configuration γ_{T+2} of e', a contradiction. ■

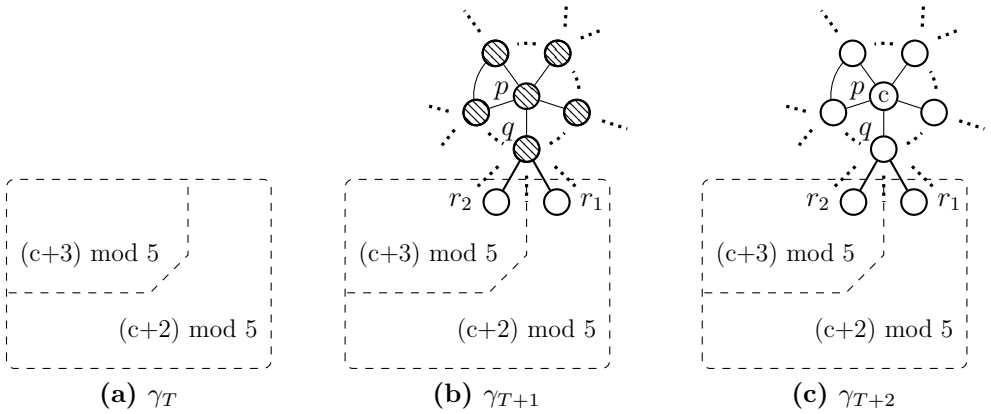
We now focus on dynamic patterns for which C is True but ULC is False and that cannot be included into ρ, unless the specification of Alg is violated. Such a pattern is defined below and will be used for the case $\alpha = 5$.

Let ζ be the dynamic pattern such that for every two graphs $\mathcal{G}_i$ and $\mathcal{G}_j$, $\zeta(\mathcal{G}_i, \mathcal{G}_j)$ if and only if the following conditions hold:

- both $\mathcal{G}_i$ and $\mathcal{G}_j$ are connected,
- $|V_i \cap V_j| \geq 2$, and
- $\exists p \in V_j \setminus V_i$ such that $\gamma_j(p).\mathcal{N} \cap V_i = \emptyset$ and $\exists q \in \gamma_j(p).\mathcal{N}, |\gamma_j(q).\mathcal{N} \cap V_i| \geq 2$.

Theorem 5.3

If $\alpha = 5$, then for every graphs $\mathcal{G}$ and $\mathcal{G}'$, we have $\zeta(\mathcal{G}, \mathcal{G}') \Rightarrow \neg\rho(\mathcal{G}, \mathcal{G}')$.



(a) γ_T **(b)** γ_{T+1} **(c)** γ_{T+2}

Figure 5.2 – Execution e' in the proof of Theorem 5.3. The hachured nodes are in bootstate. The value inside the node is the value of its clock. If there is no value, its clock value is meaningless.

Proof: We illustrate the following proof with Figure 5.2. Assume, by the contradiction, that $\alpha = 5$ but there exists two graphs $\mathcal{G}_{x-1}$ and $\mathcal{G}_x$ such that $\zeta(\mathcal{G}_{x-1}, \mathcal{G}_x)$ and $\rho(\mathcal{G}_{x-1}, \mathcal{G}_x)$.

By Lemma 5.3, there exists a configuration $\gamma_{x-1} \in \mathscr{L}_{\text{SU}}$ (*n.b.*, $\mathcal{G}_{x-1}$ is connected, by definition). Consider now the configuration γ_x of topology $\mathcal{G}_x$ such that $\gamma_{x-1} \mapsto^{d,\rho} \gamma_x$ and $\gamma_{x-1} \mapsto^{d,zeta} \gamma_x$, and no process is activated between γ_{x-1} and γ_x.

Let p and q be two nodes such that

1. $p \in V_x \setminus V_{x-1}$,
2. $q \in V_x \setminus V_{x-1}$ and $q \in \gamma_x(p).\mathcal{N}$,
3. $\forall v \in \gamma_x(p).\mathcal{N}$, $v \in V_x \setminus V_{x-1}$,
4. q has at least two neighbors r_1 and r_2 belonging to $V_x \cap V_{x-1}$

(By definition of ζ, p, q, r_1, and r_2 necessarily exist.) Then, p and its neighbors (in particular q) are in bootstate in γ_x. So, by Lemma 5.1, they all are enabled in γ_x and if they move, they will be not in bootstate and their clock values will belong to $\{0, \ldots, 4\}$ in the configuration that follows γ_x. Let c be the clock value of p in the next configuration, if p moves.

By the liveness of the strong unison and Lemma 5.2, there exists an execution e in $\mathcal{E}^0_{\text{ALG}}(\mathscr{L}_{\text{SU}})$ on the graph $\mathcal{G}_{x-1}$ which contains a configuration γ_T where r_1 and r_2 are not in bootstate and have two different clock values, one being $(c+2) \mod 5$ and the other $(c+3) \mod 5$. Without the loss of generality, assume that $\gamma_T(r_1).clock = (c+2) \mod 5$ and $\gamma_T(r_2).clock = (c+3) \mod 5$, see Figure 5.2a.

Consider now another execution $e' \in \mathcal{E}^{1,\rho}_{\text{ALG}}(\mathscr{L}_{\text{SU}})$ having a prefix common to e until γ_T. Assume that the unfair daemon introduces a ρ-dynamic step (it is possible since there was no dynamic step until now). Since $\mathcal{G}_T = \mathcal{G}_{x-1}$, the daemon can choose a step $\gamma_T \mapsto^{d,\rho} \gamma_{T+1}$, where no process moves and $\mathcal{G}_{T+1} = \mathcal{G}_x$. Now, $\forall v \in V_{T+1} \setminus V_T$, $\gamma_{T+1}(v) = \gamma_x(v)$, so again, in γ_{T+1}, p and all its neighbors (in particular q) are in bootstate and enabled. Moreover, if they move, they will not be in bootstate and their clock values will belong to $\{0, \ldots, 4\}$ in γ_{T+2}, by Lemma 5.1. Moreover, p is in the same situation as in γ_x, so if it moves, its clock is equal to c in γ_{T+2}. Then, r_1 and r_2 are both

neighbors of q which are still not in bootstate and still with clock values $(c+2) \mod 5$ and $(c+3) \mod 5$, see Figure 5.2b. By definition, since strong unison is satisfied in γ_T (by assumption), the partial unison necessarily holds all along the suffix of e' starting at γ_{T+1}. Assume that the daemon exactly selects p and its neighbors in the next static step $\gamma_{T+1} \mapsto^s \gamma_{T+2}$. In γ_{T+2} (Figure 5.2c), r_1 and r_2 are still not in bootstate and their clocks are still respectively equal to $(c+2) \mod 5$ and $(c+3) \mod 5$, since they did not move. Moreover, p is no more in bootstate and its clock equals c. Now, in γ_{T+2}, q is no more in bootstate and its clock value belongs to $\{0, \ldots, 4\}$. That clock value should differ of at most one increment (mod 5) from the clocks of p, r_1, and r_2 since partial unison holds in γ_{T+1} and all subsequent configurations. If the clock of q equals:

- c or $(c+1) \mod 5$, the difference between the clocks of q and r_2 is at least 2 increments (mod 5),
- $(c+2) \mod 5$ or $(c+3) \mod 5$, the difference between the clocks of q and p is at least 2 increments (mod 5),
- $(c+4) \mod 5$, the difference between the clocks of q and r_1 is 2 increments (mod 5).

Hence, the safety of partial unison is necessarily violated in the configuration γ_{T+2} of e', a contradiction. ■

Since for every graphs $\mathcal{G}$ and $\mathcal{G}'$, $\zeta(\mathcal{G},\mathcal{G}') \Rightarrow \mathsf{C}(\mathcal{G},\mathcal{G}') \wedge \neg\mathsf{ULC}(\mathcal{G},\mathcal{G}')$, the following corollary means that there exist dynamic patterns in ζ (hence in C but not ULC) that cannot be supported by ρ, unless Alg fails:

The previous theorem states that no ρ-dynamic step can satisfy ζ, unless Alg fails. Now, by definition, for every graphs $\mathcal{G}$ and $\mathcal{G}'$, $\zeta(\mathcal{G},\mathcal{G}') \Rightarrow \mathsf{C}(\mathcal{G},\mathcal{G}') \wedge \neg\mathsf{ULC}(\mathcal{G},\mathcal{G}')$. Hence, the following corollary holds.

Corollary 5.1

If $\alpha = 5$, then there exist graphs $\mathcal{G}$ and $\mathcal{G}'$ such that $\mathsf{C}(\mathcal{G},\mathcal{G}') \wedge \neg\mathsf{ULC}(\mathcal{G},\mathcal{G}') \wedge \neg\rho(\mathcal{G},\mathcal{G}')$.

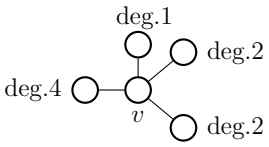
The theorem below provides the same kind of results as Corollary 5.1 for $\alpha = 4$.

Theorem 5.4

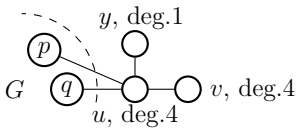
If $\alpha = 4$, then there exist graphs $\mathcal{G}$ and $\mathcal{G}'$ such that $\mathsf{C}(\mathcal{G},\mathcal{G}') \wedge \neg\mathsf{ULC}(\mathcal{G},\mathcal{G}') \wedge \neg\rho(\mathcal{G},\mathcal{G}')$.

Proof: We illustrate the following proof with Figure 5.3. Assume, by the contradiction, that $\alpha = 4$ but for every two graphs G and G' we have $\neg\mathsf{C}(\mathcal{G},\mathcal{G}') \vee \mathsf{ULC}(\mathcal{G},\mathcal{G}') \vee \rho(\mathcal{G},\mathcal{G}')$, *i.e.*, $\mathsf{C}(\mathcal{G},\mathcal{G}') \wedge \neg\mathsf{ULC}(\mathcal{G},\mathcal{G}') \Rightarrow \rho(\mathcal{G},\mathcal{G}')$.

To reduce the number of cases in the proof, we start by fixing a local proof environment, without loss of generality. To that goal, we consider a configuration γ_i in $\mathscr{L}_{\text{SU}}$ such that $\mathcal{G}_i$ is connected and contains at least one node. Consider also any ρ-dynamic step, $\gamma_i \mapsto^{d,\rho} \gamma_{i+1}$, that adds five nodes u, v, w, x and y to $\mathcal{G}_i$ in such way that the neighbors of v in $\mathcal{G}_{i+1}$ is $\{u,w,x,y\}$ and the respective degrees of u, w, x, and y are 1, 2, 2, and 4, see for instance Figure 5.3.(d). Notice that from its local point of view, v cannot distinguish configuration γ_{i+1} from any other configuration resulting from the



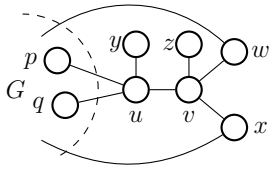
(a) After adding v and its neighbors, v is enabled and $v.clock$ is set to 0 if v moves.



(b) A dynamic step in $\mathcal{F}(G)$. Claim 0: u is enabled and the next value of $u.clock$ is completely determined by the local states of p and q.

	$p.clock$	$q.clock$	$u.clock$
Claim 1	2	3	3
Claim 2	1	2	1
Claim 3	2	2	1 or 3

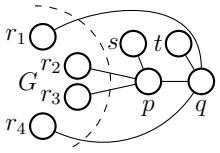
(c) Claim 1, 2, 3: after adding u, v, y, u is enabled and $u.clock$ will be fixed by $p.clock$ and $q.clock$



(d) Proof of Claim 1, 2, 3

	γ_x	γ_{x+1}		γ_y	γ_{y+1}		γ_z	γ_{z+1}
r_1	**1**	2		2	**2**		**2**	3
r_4	1	**2**		2	**2**		2	**3**
	1	2	$\cdots$	2	2	$\cdots$	2	3
	$\vdots$	$\vdots$		$\vdots$	$\vdots$		$\vdots$	$\vdots$
r_2	1	**2**		**2**	2		**2**	3
r_3	1	**2**		**2**	2		**2**	3
p		1	$\cdots$	1	3		3	

(e) Claim 3, 4, 5, and end of the proof.



(f) Proof of Claim 3.

Figure 5.3 – Proof of Theorem 5.4. The hachured nodes are in bootstate. The value inside a node is its clock value. If no value is given, then the clock value is meaningless. Notation "deg." stands for degree.

addition of v and its neighbors, since v and all its neighbors are in bootstate. In γ_{i+1}, due to Lemma 5.1, v is enabled and if v moves, then $v.clock \in \{0, ..., 3\}$ in the next configuration. Without the loss of generality, we fix the value to 0. Hence, follows.

Local Proof Environment: Let v be a node surrounded by 4 neighbors having respectively degree 1, 2, 2 and 4 such that v and its neighbors are all in *bootstate*. In such a configuration, v is enabled and if v moves in the next step, then v sets $v.clock$ to 0. See Figure 5.3.(a). □

Let $\mathcal{G} = (V, E)$ be any connected graph of at least two nodes, p, q. Let $\mathcal{F}(\mathcal{G})$ be the family of graphs $\mathcal{G}' = (V', E')$ obtained by applying a dynamic step on $\mathcal{G}$ such that

1. $\mathsf{C}(\mathcal{G}, \mathcal{G}')$ holds,
2. $V \subseteq V'$,
3. $\{u, y, v\} \subseteq V' \setminus V$, and
4. E' contains at least all links in E plus the following additional links.

- y is a 1-degree node linked to u;
- u has degree 4, it is linked to y, v, and two nodes of V; and
- v has degree 4 and is, in particular, linked to u.

See Figure 5.3.(b). Notice that for every $\mathcal{G}'$ in $\mathcal{F}(\mathcal{G})$, $\neg\mathsf{ULC}(\mathcal{G},\mathcal{G}')$ holds, due to node y. Hence, any dynamic step that transforms $\mathcal{G}$ into $\mathcal{G}'$ is a ρ-dynamic step.

Let $\gamma \in \mathscr{L}_{\mathrm{SU}}$ whose topology is $\mathcal{G}$. Let $\gamma \mapsto^{d,\rho} \gamma'$ be a ρ-dynamic step where no process executes and that transform $\mathcal{G}$ into $\mathcal{G}' \in \mathcal{F}(\mathcal{G})$.

Claim 0: In γ', process u is enabled (by Lemma 5.1) and, if it executes, the new value of $u.clock$ is completely determined by $\gamma(p)$ and $\gamma(q)$.

Claim 1: If p and q respectively have clock value 2 and 3 in configuration γ, then for every $G' \in \mathcal{F}(\mathcal{G})$, u is enabled in γ' and if it executes, $u.clock$ has value 3 in next configuration.

Proof of the claim: By Claim 0, u is enabled at γ' and its next clock value is fully determined by $\gamma(p)$ and $\gamma(q)$ whatever the graph G' of $\mathcal{F}(\mathcal{G})$. So, to determine this value, it is sufficient to compute it from a particular graph $\mathcal{G}'$ of $\mathcal{F}(\mathcal{G})$. We build this graph as follows: $V \subseteq V'$, $\{u,v,w,x,y,z\} \subseteq V' \setminus V$, and E' contains all links in E plus the following additional links.

- u has four neighbors: v, y, p and q,
- v has four neighbors: u, w, x, and z,
- w has two neighbors: v and a node in V,
- x has two neighbors: v and a node in V, and
- y and z have degree one.

See Figure 5.3.(d). Notice that, by definition, $\mathcal{G}' \in \mathcal{F}(\mathcal{G})$.

We consider γ as the first configuration of an execution in $\mathcal{E}^{1,\rho}_{\mathscr{L}_{\mathrm{SU}}}$. Then, the first step of the execution is the step $\gamma \mapsto^{d,\rho} \gamma'$. Hence in γ', u and v are both enabled (see the local proof environment and Claim 0): assume that the daemon exactly selects u and v for next static step $\gamma' \mapsto^{s} \gamma''$. In γ'', the states of p and q have not changed, v and u are no more in bootstate and $v.clock = 0$, from the local proof environment. The clock value $u.clock$ should differ from the clocks of p, q, and v by at most one increment (mod 4) since partial unison holds in γ' and γ''. So, u necessarily has clock value 3 in γ''. □

Using a similar reasoning, we obtain the following two claims.

Claim 2: If p and q respectively have clock value 1 and 2 at configuration γ, then for every $\mathcal{G}' \in \mathcal{F}(\mathcal{G})$, u is enabled in γ' and if it executes, $u.clock$ has value 1 in next configuration.

Claim 3: If p and q have both clock value 2 in configuration γ, then for every $\mathcal{G}' \in \mathcal{F}(\mathcal{G})$, u is enabled in γ' and if it executes, $u.clock$ has value either 1 or 3 in next configuration.

Consider now any regular[3] connected graph $\mathcal{G} = (V,E)$ of at least four nodes, r_1, r_2, r_3 and r_4. Let $e = (\gamma_i)_{i\geq 0} \in \mathcal{E}^{0,\rho}_{\mathscr{L}_{\mathrm{SU}}}$ be a *synchronous* execution of the algorithm on graph $\mathcal{G}$, such that in γ_0 every process has exactly same state. As the execution is synchronous, the algorithm deterministic, and the graph regular, this property is invariant all along the execution: in every configuration γ_i of e, $\forall p \in V, \gamma_i(p) = \gamma_i(r_1)$.

[3]Regular means that all nodes have the same degree.

Now, by hypothesis, there exists a configuration γ_i in e such that $\gamma_i \in \mathscr{L}_{\text{SU}}$. From γ_i, every clock in the graph regularly increments (modulo 4). We denote by $\gamma_x \mapsto^s \gamma_{x+1}$ some step in e with $x > i$ such that clock value increments from 1 in γ_x to 2 in γ_{x+1}. Moreover, let $\gamma_z \mapsto^s \gamma_{z+1}$ the next step in e where clocks increment again, namely clocks increment from 2 in γ_z to 3 in γ_{z+1}. See Figure 5.3.(e). Notice that in each configuration between γ_x (included) and γ_{z+1} (included) every process has the same state. Moreover, in all configurations between γ_{x+1} (included) and γ_z (included) all processes have clock value 2.

For every $k \in \{x+1, ..., z\}$, we build the execution $e_k \in \mathcal{E}^{1,\rho}_{\mathscr{L}_{\text{SU}}}$ such that e and e_k have the same prefix $\gamma_0, ..., \gamma_k$. But, in γ_k, e_k suffers from a dynamic step $\gamma_k \mapsto_d \gamma'_k$ built as follows: no process executes, no node or edge disappears, yet the pattern of Figure 5.3.(f) is added; namely this step built a graph $\mathcal{G}' = (V', E')$ such that $V \subseteq V'$, $V' \setminus V = \{p, q, s, t\}$, E' contains all links of E plus the following additional links.

- p has four neighbors: r_2, r_3, q, s,
- q has four neighbors: r_1, r_4, p, t,
- s has one neighbor: p, and
- t has one neighbor: q.

Again, notice that, by definition, $\mathcal{G}' \in \mathcal{F}(\mathcal{G})$. Hence any dynamic step that transforms $\mathcal{G}$ into $\mathcal{G}'$ is a ρ-dynamic step.

For every $k \in \{x+1, ..., z\}$, p and q are enabled in γ'_k (by Lemma 5.1). We note $\gamma'_k \mapsto^s \gamma''_k$ the next static step after the dynamic step where p and q are the only nodes activated by the daemon. In the following, we are interesting in the value of $\gamma''_k(p)$ for $k \in \{x+1, ..., z\}$. Note that for all those values, Claim 3 applies, hence $\gamma''_k(p)$ is either 1 or 3.

Claim 4: $\gamma''_{x+1}(p) = 1$

Proof of the claim: We consider the execution $e'_{x+1} \in \mathcal{E}^{1,\rho}_{\mathscr{L}_{\text{SU}}}$ with prefix $\gamma_0, ..., \gamma_x$. In γ_x, we introduce a non-synchronous static step $\gamma_x \mapsto^s \vartheta_{x+1}$ such that every process but r_1 are activated. Hence $\vartheta_{x+1}(r_1) = \gamma_x(r_1)$ (in particular the clock is 1) and $\vartheta_{x+1}(n) = \gamma_{x+1}(n)$ for every $n \neq r_1$ (with in particular a clock value equal to 2). The next step of e'_{x+1} is a ρ-dynamic step $\vartheta_{x+1} \mapsto^{d,\rho} \vartheta'_{x+1}$ that transforms G into G' and activates no process; again p and q are enabled in ϑ'_{x+1} and the next step is a static step $\vartheta'_{x+1} \mapsto^s \vartheta''_{x+1}$ where the daemon uniquely activates p and q. Claim 2 applies to q: q is enabled, and $\vartheta''_{x+1}(q).clock = 1$. Claim 3 applies to p and $\vartheta''_{x+1}(p).clock$ is either 1 or 3. Hence, to satisfy the partial unison in ϑ''_{x+1}, $\vartheta''_{x+1}(p).clock$ is necessarily equal to 1.

Now, back to execution e_{x+1}, Claim 0 applies to p in step $\gamma'_{x+1} \mapsto^s \gamma''_{x+1}$: $\gamma''_{x+1}(p)$ is fully determined by $\gamma_{x+1}(r_2)$ and $\gamma_{x+1}(r_3)$. As $\gamma_{x+1}(r_2)$ (respectively, $\gamma_{x+1}(r_3)$) has been obtained by executing the local algorithm of r_2 (respectively, r_3) and as $\vartheta'_{x+1}(r_2)$ (respectively, $\vartheta'_{x+1}(r_3)$) has been obtained exactly the same way, they are equal. Hence $\vartheta''_{x+1}(p) = \gamma''_{x+1}(p) = 1$. □

Claim 5: $\gamma''_z(p) = 3$.

Proof of the claim: We consider the execution $e'_z \in \mathcal{E}^{1,\rho}_{\mathscr{L}_{\text{SU}}}$ with prefix $\gamma_0, ..., \gamma_z$. At γ_z, we introduce a non-synchronous static step $\gamma_z \mapsto^s \vartheta_{z+1}$ such that process r_4 only has been activated. Hence $\vartheta_{z+1}(r_4) = \gamma_{z+1}(r_4)$ (in particular the clock is 3) and $\vartheta_{x+1}(n) = \gamma_z(n)$ for every $n \neq r_4$ (with clocks equal to 2). The next step of

e'_z is a ρ-dynamic step $\vartheta_{z+1} \mapsto^{d,\rho} \vartheta'_{z+1}$ that transforms $\mathcal{G}$ into $\mathcal{G}'$ and activates no process; again p and q are enabled at ϑ'_{z+1} and next step is a static step $\vartheta'_{z+1} \mapsto^{s} \vartheta''_{z+1}$ where the daemon uniquely activates p and q. Claim 1 applies to q: q is enabled, hence $\vartheta''_{z+1}(q).clock = 1$. Claim 3 applies to p and $\vartheta''_{z+1}(p).clock$ is either 1 or 3. Hence, to satisfy the partial unison in ϑ''_{z+1}, $\vartheta''_{z+1}(p).clock$ is necessarily equal to 1.

Now, back to execution e_z, Claim 0 applies to p in step $\gamma'_z \mapsto^{s} \gamma''_z$: $\gamma''_z(p)$ is fully determined by $\gamma_z(r_2)$ and $\gamma_z(r_3)$. As $\gamma_z(r_2)$ (respectively, $\gamma_z(r_3)$) has been obtained by executing the local algorithm of r_2 (respectively, r_3) and as $\vartheta'_{z+1}(r_2)$ (respectively, $\vartheta'_{z+1}(r_3)$) has been obtained exactly the same way, they are equal. Hence $\vartheta''_{z+1}(p) = \gamma''_{z+1}(p) = 3$. □

By Claims 3-5, the sequence of values $(\gamma''_i(p).clock)_{i\in\{x+1,\ldots,z\}}$ is only consists of values 1 and 3, starting with 1 and ending with 3. Hence, there exists an index $y \in \{x+1, z-1\}$ for which the value switches from 1 to 3, *i.e.*, $\gamma''_y(p) = 1$ and $\gamma''_{y+1}(p) = 3$.

We consider the execution $e'_y \in \mathcal{E}^{1,\rho}_{\mathscr{L}_{\mathrm{SU}}}$ with prefix $\gamma_0, \ldots, \gamma_y$. In γ_y, we introduce a non-synchronous static step $\gamma_y \mapsto^{s} \vartheta_{y+1}$ such that all processes are activated, except r_2 and r_3. Hence, $\vartheta_{y+1}(r_2) = \gamma_y(r_2)$, $\vartheta_{y+1}(r_3) = \gamma_y(r_3)$, and $\vartheta_{y+1}(r) = \gamma_{y+1}(r)$ for every $r \notin \{r_2, r_3\}$. The next step of e'_y is a ρ-dynamic step $\vartheta_{y+1} \mapsto^{d,\rho} \vartheta'_{y+1}$ that transforms $\mathcal{G}$ into $\mathcal{G}'$ and activates no process; again p and q are enabled in ϑ'_{y+1} and the next step is a static step $\vartheta'_{y+1} \mapsto^{s} \vartheta''_{y+1}$ where the daemon uniquely activates p and q.

Claim 0 applies to p (respectively, q) in step $\vartheta'_{y+1} \mapsto^{s} \vartheta''_{y+1}$: $\vartheta''_{y+1}(p)$ is fully determined by $\vartheta_{y+1}(r_2) = \gamma_y(r_2)$ and $\vartheta_{y+1}(r_3) = \gamma_y(r_3)$ (respectively, $\vartheta_{y+1}(r_1) = \gamma_{y+1}(r_1)$ and $\vartheta_{y+1}(r_1) = \gamma_{y+1}(r_1)$). Hence, $\vartheta''_{y+1}(p) = \gamma''_y(p) = 1$ and $\vartheta''_{y+1}(q) = \gamma''_y(p) = 3$. This contradicts the fact that the partial unison holds in γ''_y. ■

5.5 Self-Stabilizing Strong Unison

In this section, we propose an algorithm, called $\mathcal{SU}$, which is self-stabilizing for the strong unison problem in any arbitrary connected anonymous network. This algorithm works for any period $\alpha \geq 2$ (recall that the problem is undefined for $\alpha < 2$) and is based on an algorithm previously proposed by Boulinier in [Bou07]. This latter is self-stabilizing for the weak unison problem and works for any period $\beta > n^2$, where n is the number of processes. We first recall the algorithm of Boulinier, called here Algorithm $\mathcal{WU}$, in Subsection 5.5.1. Notice that the notation used in this algorithm will be also applicable to our algorithms. We present Algorithm $\mathcal{SU}$, its proof of correctness, and its complexity analysis in Subsection 5.5.2. Algorithms $\mathcal{WU}$ and $\mathcal{SU}$ being only self-stabilizing, all their executions contain no topological change, yet start from arbitrary configurations. Consequently, the topology of the network consists in a connected graph $G = (V, E)$ of n nodes which is fixed all along the execution.[4] Moreover, no bootstate has to be defined. Recall that $\mathcal{D}$ is the diameter of G.

[4]Precisely, for both $\mathcal{WU}$ and $\mathcal{SU}$, we have $\forall \gamma_i \in \mathcal{C}$, $G_i = G$.

Algorithm 10 – Actions of Process p in Algorithm $\mathcal{WU}$.

Inputs.

- $\beta \in \mathbb{N}$ such that $\beta > n^2$
- $\mu \in \mathbb{N}$ such that $n \leq \mu < \frac{\beta}{2}$

Variables.

- $p.t \in \{0, \ldots, \beta - 1\}$

Actions.

$\mathcal{WU}$-**N**	::	$\forall q \in \mathcal{N}p, p.t \preceq_{\beta,\mu}$	$\rightarrow$	$p.t := (p.t + 1) \mod \beta$
$\mathcal{WU}$-**R**	::	$\exists q \in \mathcal{N}p, d_\beta(p.t, q.t) > \mu \vee p.t \neq 0$	$\rightarrow$	$p.t := 0$

5.5.1 Algorithm $\mathcal{WU}$

Algorithm $\mathcal{WU}$, see Algorithm 10 for its formal code, has been proposed by Boulinier in his PhD thesis [Bou07]. Actually, it is a generalization of the self-stabilizing weak unison algorithm proposed by Couvreur et al. [CFG92]. In Algorithm $\mathcal{WU}$, each process p is endowed with a clock variable $p.t \in \{0, \ldots, \beta - 1\}$, where β is its period. β should be greater than n^2. The algorithm also uses another constant, noted μ, which should satisfy $n \leq \mu < \frac{\beta}{2}$.

Notations. We define the *delay* between two integer values x and y by the function $d_\beta(x, y) = \min\big((x-y) \mod \beta, (y-x) \mod \beta\big)$. Then, let $\preceq_{\beta,\mu}$ be the relation such that for every two integer values x and y, $x \preceq_{\beta,\mu} y \equiv \big((y - x) \mod \beta\big) \leq \mu$. (*N.b.*, $\preceq_{\beta,\mu}$ is only defined for $\mu < \frac{\beta}{2}$, see Definition 14 in [Bou07].)

Overview of $\mathcal{WU}$. Two actions are used to maintain the clock $p.t$ at each process p. When the delay between $p.t$ and the clocks of some neighbors is greater than one, but the maximum delay is not too big (that is, does not exceed μ), then it is possible to "normally" converge, using $\mathcal{WU}$-**N**-action, to a configuration where the delay between those clocks is at most one by incrementing the clocks of the most behind processes among p and its neighbors. Moreover, once legitimacy is achieved, p can "normally" increment its clock still using $\mathcal{WU}$-**N**-action when it is on time or one increment late with all its neighbors. In contrast, if the delay is too big (that is, the delay between the clocks of p and one of its neighbors is more than μ) and the clock of p is not yet reset, then p should reset its clock to 0 using $\mathcal{WU}$-**R**-action.

From [Bou07], we have the following theorem.

Theorem 5.5

Algorithm $\mathcal{WU}$ is self-stabilizing for SP_{WU} (the specification of weak unison) in an arbitrary connected network assuming a distributed unfair daemon. Its set of legitimate configurations is

$$\mathscr{L}_{\text{WU}} = \{\gamma \in \mathcal{C} : \forall p \in V, \forall q \in \mathcal{N}\gamma(p), d_\beta(\gamma(p).t, \gamma(q).t \leq 1\}$$

Its stabilization time is at most $n + \mu\mathcal{D}$ rounds, where n (resp. $\mathcal{D}$) is the size (resp. diameter) of the network and μ is a parameter satisfying $n \leq \mu < \frac{\beta}{2}$.

By definition, $\mathcal{D} < n$, consequently we have:

Remark 5.2

Once Algorithm $\mathcal{WU}$ has stabilized, the delay between t-clocks of any two arbitrary far processes is at most $n-1$.

Some other useful results from [Bou07] about Algorithm $\mathcal{WU}$ are recalled below.

Results from [Bou07]. Algorithm $\mathcal{WU}$ is an instance of the parametric algorithm GAU in [Bou07]: $\mathcal{WU} = GAU(\beta, 0, \mu)$. The following five lemmas (5.4-5.8) are used to establish the self-stabilization of $\mathcal{WU}$ for SP_{WU} using the set of legitimate configurations $\mathscr{L}_{\text{WU}}$. The proof of self-stabilization is actually divided into several steps. The first step (Lemma 5.5) consists in showing the convergence of $\mathcal{WU}$ from $\mathcal{C}_{\mathcal{WU}}$ to C_μ, where C_μ is the set of configurations where the delay between the clocks of two neighbors is at most μ, *i.e.*

$$C_\mu = \{\gamma \in \mathcal{C} : \forall p \in V, \forall q \in \gamma(p).\mathcal{N}, d_\beta(\gamma(p).t, \gamma(q).t) \leq \mu\}$$

C_μ is shown to be closed under $\mathcal{WU}$ in Lemma 5.4. (Notice that $\mathscr{L}_{\text{WU}} \subseteq C_\mu$.) The liveness part of SP_{WU}(the clock $p.t$ of every process p goes through each value in $\{0, \ldots, \beta-1\}$ in increasing order infinitely often) is shown for every execution starting from C_μ in Lemma 5.6.

Lemma 5.4 (*Property 8 in [Bou07]*)

C_μ is closed under $\mathcal{WU}$.

Lemma 5.5 (*Theorem 56 in [Bou07]*)

If $n \leq \mu < \frac{\beta}{2}$, then $\forall e \in \mathcal{E}^0_{\mathcal{WU}}$, $\exists \gamma \in e$ such that $\gamma \in C_\mu$.

Lemma 5.6 (*Theorem 21 in [Bou07]*)

If $\beta > n^2$, then $\forall e \in \mathcal{E}^0_{\mathcal{WU}}(C_\mu)$, e satisfies the liveness part of SP_{WU}.

Then, the second step consists of showing closure of $\mathscr{L}_{\text{WU}}$ under $\mathcal{WU}$ (Lemma 5.7) and the convergence from C_μ to $\mathscr{L}_{\text{WU}}$ (Lemma 5.8). Regarding the correctness, the safety part of SP_{WU} is ensured by definition of $\mathscr{L}_{\text{WU}}$, whereas the liveness part is already ensured by Lemma 5.6. Precisely:

Lemma 5.7 (*Property 2 in [Bou07]*)

$\mathscr{L}_{\text{WU}}$ is closed under $\mathcal{WU}$.

Lemma 5.8 (*Theorems 29 in [Bou07]*)

If $\beta > n^2$ and $\mu < \frac{\beta}{2}$, then $\forall e \in \mathcal{E}^0_{\mathcal{WU}}(C_\mu)$, $\exists \gamma \in e$ such that $\gamma \in \mathscr{L}_{\text{WU}}$.

Some performances of Algorithm $\mathcal{WU}$ are recalled in Theorems 5.6 and 5.7 below.

Theorem 5.6 (*Theorem 61 in [Bou07]*)

If $n \leq \mu < \frac{\beta}{2}$, the convergence time of $\mathcal{WU}$ from $\mathcal{C}_{\mathcal{WU}}$ to C_μ is at most n rounds.

Theorem 5.7 (*Theorems 20 and 28 in [Bou07]*)

If $\beta > n^2$ and $\mu < \frac{\beta}{2}$, the convergence time of $\mathcal{WU}$ from C_μ to $\mathscr{L}_{\text{WU}}$ is at most $\mu\mathcal{D}$ rounds.

Finally, Lemma 5.9 below is a technical result about the values of t-variables.

Lemma 5.9 (*Theo. 20, Lem. 22, Propos. 25, and Property 27 in [Bou07]*)

If $\beta > n^2$ and $\beta > 2\mu$, then $\forall e = (\gamma_i)_{i\geq 0} \in \mathcal{E}^0_{\mathcal{WU}}(C_\mu)$, there exists a so-called shifting function *$f : \mathcal{C} \times V \to \mathbb{Z}$ such that $\forall i \geq 0$, $\forall p, q \in V$,*

- *$\forall 0 \leq i \leq j$, $f(\gamma_i, p) \leq f(\gamma_j, p)$,*
- *$p.t \preceq_{\beta,\mu} q.t$ if and only if $f(\gamma_i, p) \leq f(\gamma_i, q)$,*
- *$\forall i \geq 0$, $\forall p \in V$, $f(\gamma_i, p) \mod \beta = \gamma_i(p).t$, and*
- *$|f(\gamma_i, p) - f(\gamma_i, q)| = d_\beta(\gamma_i(p).t, \gamma_i(q).t)$.*

Complexity Analysis. Let C_μ be the set of configurations where the delay between two neighboring clocks is at most μ. Below, we prove in Lemma 5.10 (resp. Lemma 5.11) a bound on the time required to ensure that all t-variables have incremented k times. This bound holds since the system has reached a configuration of C_μ (resp. $\mathscr{L}_{\text{WU}}$).

Lemma 5.10

*$\forall k \geq 1$, $\forall e \in \mathcal{E}^0_{\mathcal{WU}}(C_\mu)$, every process p increments $p.t$ executing $\mathcal{WU}$-**N**-action at least k times every $\mu\mathcal{D} + k$ rounds, where $\mathcal{D}$ is the diameter of the network.*

Proof: Let $k \geq 1$. Let $e = (\gamma_i)_{i\geq 0} \in \mathcal{E}^0_{\mathcal{WU}}(C_\mu)$. Using Lemma 5.9, there is a shifting function f such that $\forall i \geq 0$, $\forall p, q \in V$, $|f(\gamma_i, p) - f(\gamma_i, q)| \leq \mu\mathcal{D}$. For every $i \geq 0$, we note $f^{\min}_{\gamma_i} = \min\{f(\gamma_i, x) : x \in V\}$. $\mathcal{WU}$-**N**-action is enabled in γ_i at every process $x \in V$ for which $f(\gamma_i, x) = f^{\min}_{\gamma_i}$. So, after one round, every such a process x has incremented its t-variable (by $\mathcal{WU}$-**N**) at least once. Let γ_j be the first configuration after one round. Then, $f^{\min}_{\gamma_j} \geq f^{\min}_{\gamma_i} + 1$. We now consider γ_d to be the first configuration after $\mu\mathcal{D} + k$ rounds, starting from γ_i. Using the same arguments as for γ_j inductively, we have $f^{\min}_{\gamma_d} \geq f^{\min}_{\gamma_i} + \mu\mathcal{D} + k$ $(*)$.

Let p be a process in V. By definitions of f and $f^{\min}_{\gamma_i}$, we have that $f^{\min}_{\gamma_i} \leq f(\gamma_i, p) \leq f^{\min}_{\gamma_i} + \mu\mathcal{D}$ $(**)$. Assume now that p increments $\sharp incr < k$ times p.t between γ_i and γ_d. Then,

$$\begin{array}{rcl} f(\gamma_d, p) = f(\gamma_i, p) + \#incr & < & f(\gamma_i, p) + k \text{ (assumption on } \#incr) \\ & \leq & f^{\min}_{\gamma_i} + \mu\mathcal{D} + k, \text{ by } (**) \\ & \leq & f^{\min}_{\gamma_d}, \text{ by } (*) \end{array}$$

So, p satisfies $f(\gamma_d, p) < f^{\min}_{\gamma_d}$, a contradiction. ■

Lemma 5.11

*$\forall k \geq 1$, $\forall e \in \mathcal{E}^0_{\mathcal{WU}}(\mathscr{L}_{\text{WU}})$, every process p increments its clock $p.t$ executing $\mathcal{WU}$-**N**-action at least k times every $\mathcal{D} + k$ rounds, where $\mathcal{D}$ is the diameter of the network.*

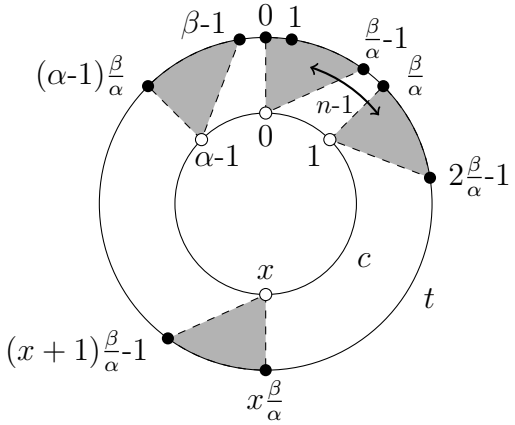


Figure 5.4 – Relationship between variables t and c.

Proof: The proof of this lemma is exactly the same as the one of Lemma 5.10, yet replacing C_μ by $\mathscr{L}_{\text{WU}}$ and $\mu\mathcal{D}$ by $\mathcal{D}$. ■

5.5.2 Algorithm $\mathcal{SU}$

In this subsection, we still assume a non-dynamic context (no topological change) and we use the notations defined in Subsection 5.5.1. Algorithm $\mathcal{SU}$ is a straightforward adaptation of Algorithm $\mathcal{WU}$. More precisely, Algorithm $\mathcal{SU}$ maintains two clocks at each process p. The first one, $p.t \in \{0, \ldots, \beta - 1\}$, is called the *internal clock* and is maintained exactly as in Algorithm $\mathcal{WU}$. Then, $p.t$ is used as an internal pulse machine to increment a second, yet actual, clock of Algorithm $\mathcal{SU}$ $p.c \in \{0, \ldots, \alpha - 1\}$, also referred to as *external clock*.

Algorithm $\mathcal{SU}$ (see Algorithm 11), is designed for any period $\alpha \geq 2$. Its actions $\mathcal{SU}$**-N** and $\mathcal{SU}$**-R** are identical to actions $\mathcal{WU}$**-N** and $\mathcal{WU}$**-R** of Algorithm $\mathcal{WU}$, except that we add the computation of the external c-clock in their respective statement.

We already know that Algorithm $\mathcal{WU}$ stabilizes to a configuration from which t-clocks regularly increment while preserving a bounded delay of at most one between two neighboring processes, and so of at most $n-1$ between any two processes (see Remark 5.2). Algorithm $\mathcal{SU}$ implements the same mechanism to maintain $p.t$ at each process p and computes $p.c$ from $p.t$ as a normalization operation from clock values in $\{0, \ldots, \beta - 1\}$ to $\{0, \ldots, \alpha - 1\}$: each time the value of $p.t$ is modified, $p.c$ is updated to $\left\lfloor \frac{\alpha}{\beta} p.t \right\rfloor$. Hence, we can set β in such way that $K = \frac{\beta}{\alpha}$ is greater than or equal to n (here, we chose $K > \mu \geq n$ for sake of simplicity) to ensure that, when the delay between any two t-clocks is at most $n - 1$, the delay between any two c-clocks is at most one, see Figure 5.4. Furthermore, the liveness of $\mathcal{WU}$ ensures that every t-clock increments infinitely often, hence so do c-clocks.

Remark 5.3

Since $\beta > \mu^2$ and $\mu \geq 2$, we have $\beta \geq 2\mu$.

Algorithm 11 – Actions of Process p in Algorithm $\mathcal{SU}$.

Inputs.

- $\alpha \in \mathbb{N}$ such that $\alpha \geq 2$
- $\mu \in \mathbb{N}$ such that $\mu \geq \max(n, 2)$
- $\beta \in \mathbb{N}$ such that $\beta > \mu^2$ and $\exists K$ such that $K > \mu$ and $\beta = K\alpha$

Variables.

- $p.c \in \{0, \ldots, \alpha - 1\}$
- $p.t \in \{0, \ldots, \beta - 1\}$

Actions.

$\mathcal{SU}$-**N**	::	$\forall q \in \mathcal{N}p, p.t \preceq_{\beta,\mu}$	$\rightarrow$	$p.t := (p.t + 1) \mod \beta$ $p.c := \left\lfloor \frac{\alpha}{\beta} p.t \right\rfloor$
$\mathcal{SU}$-**R**	::	$\exists q \in \mathcal{N}p, d_\beta(p.t, q.t) > \mu \vee p.t \neq 0$	$\rightarrow$	$p.t := 0$ $p.c := 0$

Remark 5.4

By construction and from Remark 5.3, all results on t-clocks in Algorithm $\mathcal{WU}$ also holds for t-clocks in Algorithm $\mathcal{SU}$.

Theorem 5.8 below states that Algorithm $\mathcal{SU}$ is self-stabilizing for the strong unison problem. We detail the proof of this intuitive result in the sequel.

Theorem 5.8

Algorithm $\mathcal{SU}$ is self-stabilizing for SP_{SU} (the specification of the strong unison) in any arbitrary connected anonymous network assuming a distributed unfair daemon. Its stabilization time is at most $n + (\mu + 1)\mathcal{D} + 1$ rounds, where n (resp. $\mathcal{D}$) is the size (resp. diameter) of the network and μ is a parameter satisfying $\mu \geq \max(n, 2)$.

Correctness. We first define a set of legitimate configurations w.r.t. specification SP_{SU} (Definition 5.9). Then, we prove the closure and convergence w.r.t. those legitimate configurations (see Lemmas 5.12 and 5.13). Afterwards, we prove the correctness w.r.t. specification SP_{SU} in any execution starting in a legitimate configuration, namely, safety is shown in Lemma 5.16 and liveness is proven in Lemma 5.17.

Definition 5.9 (*Legitimate Configurations of $\mathcal{SU}$* w.r.t. SP_{SU})

A configuration γ of $\mathcal{SU}$ is *legitimate* w.r.t. SP_{SU} if and only if

1. $\forall p \in V$, $\forall q \in \gamma(p).\mathcal{N}$, $d_\beta(\gamma(p).t, \gamma(q).t) \leq 1$.
2. $\forall p \in V$, $\gamma(p).c = \left\lfloor \frac{\alpha}{\beta} \gamma(p).t \right\rfloor$.

We denote by $\mathscr{L}_{\text{SU}}$ the set of legitimate configurations of $\mathcal{SU}$ w.r.t. SP_{SU}.

By definition, $\mu \geq n > 0$, hence from Definition 5.9, follows.

Remark 5.5

In any legitimate configuration $\gamma \in \mathscr{L}_{\text{SU}}$, $\forall p, q \in V$, $d_\beta(\gamma(p).t, \gamma(q).t) \leq \mu$.

Lemma 5.12 (*Closure*)

$\mathscr{L}_{\text{SU}}$ is closed under $\mathcal{SU}$.

Proof: First, from Theorem 5.5 and Remark 5.4, note that the set of legitimate configurations defined for $\mathcal{WU}$ is also closed for $\mathcal{SU}$. Hence we only have to check closure for the second constraint of Definition 5.9, the one on c-variables.

Let $\gamma \in \mathscr{L}_{\text{SU}}$ be a legitimate configuration of $\mathcal{SU}$ and let $\gamma \mapsto^s \gamma'$ be a static step of $\mathcal{SU}$. Let $p \in V$. As $\gamma \in \mathscr{L}_{\text{SU}}$, $\gamma(p).c = \left\lfloor \frac{\alpha}{\beta}\gamma(p).t \right\rfloor$. Either p does not execute any action during step $\gamma \mapsto^s \gamma'$, or p executes $\mathcal{SU}$-**N** or $\mathcal{SU}$-**R**-action. These two actions update $p.c$ according to the new value of $p.t$. Hence, $\gamma'(p).c = \left\lfloor \frac{\alpha}{\beta}\gamma'(p).t \right\rfloor$. ■

Lemma 5.13 (*Convergence*)

$\mathcal{C}$ (the set of all possible configurations) converges to $\mathscr{L}_{\text{SU}}$ under $\mathcal{SU}$.

Proof: From Theorem 5.5 and Remark 5.4, the set of legitimate configurations for $\mathcal{WU}$ is also reached in a finite number of steps for $\mathcal{SU}$. Hence, we only have to check that the second constraint (the one on c-variables) is also achievable within a finite number of steps.

Again by Theorem 5.5 and Remark 5.4, liveness of Specification SP_{WU} is ensured by $\mathcal{WU}$ and therefore by $\mathcal{SU}$. Hence, after stabilization, each process p updates its internal clock $p.t$ within a finite time; meanwhile $p.c$ is also updated to $\left\lfloor \frac{\alpha}{\beta}p.t \right\rfloor$. ■

Remarks 5.6, 5.7 and Lemma 5.14 are technical results on the values of t- and c-variables that will be used to prove that the safety of Specification SP_{SU} is achieved in any execution that starts from a legitimate configuration. For all these lemmas, we assume that α, β, K are positive numbers that satisfies the constraint declared on the **Inputs** section of Algorithm $\mathcal{SU}$, namely $\beta = K\alpha$.

Remark 5.6

Let $x \in \{0, \ldots, \alpha - 1\}$ and $\xi \in \left\{0, \ldots, \frac{\beta}{\alpha} - 1\right\}$. The following equality holds:

$$\left\lfloor \frac{\alpha}{\beta}\left(x\frac{\beta}{\alpha} + \xi\right) \right\rfloor = x$$

Remark 5.7

Let $x_1, x_2 \in \{0, \ldots, \alpha - 1\}$ and $\xi_1, \xi_2 \in \left\{0, \ldots, \frac{\beta}{\alpha} - 1\right\}$. The following assertion holds: $x_1\frac{\beta}{\alpha} + \xi_1 \leq x_2\frac{\beta}{\alpha} + \xi_2 \Rightarrow x_1 \leq x_2$

We apply Remarks 5.6 and 5.7 by instantiating the value of the internal clock $t$with $x\frac{\beta}{\alpha} + \xi$. Since the value of the external clock $c$is computed as $\left\lfloor \frac{\alpha}{\beta}t \right\rfloor$ in Algorithm 11, we have $c = x$. Now, if we chose β (period of internal clocks) such that it can be written as

$\beta = K\alpha$ with K a positive integer, the value of $c = \left\lfloor \frac{\alpha}{\beta} t \right\rfloor$ is always a non negative integer which evolves according to $t = c\frac{\beta}{\alpha} + \xi$ as shown in Figure 5.4 (p. 149).

Lemma 5.14

Let $t_1, t_2 \in \{0, ..., \beta - 1\}$. The following assertion holds:

$$\forall d < K, \;\; d_\beta(t_1, t_2) \leq d \Rightarrow d_\alpha\left(\left\lfloor \frac{\alpha}{\beta} t_1 \right\rfloor, \left\lfloor \frac{\alpha}{\beta} t_2 \right\rfloor\right) \leq 1$$

Proof: Let $t_1, t_2 \in \{0, ..., \beta - 1\}$ such that $d_\beta(t_1, t_2) \leq d$. Recall that $K = \frac{\beta}{\alpha}$. We write t_1 and t_2 as $t_1 = x_1 K + \xi_1$ and $t_2 = x_2 K + \xi_2$ where $x_1, x_2 \in \{0, \ldots, \alpha - 1\}$ (resp. $\xi_1, \xi_2 \in \{0, \ldots, K - 1\}$) are the quotients (resp. remainders) of the Euclidean division of t_1, t_2 by K. From Remark 5.6, we have that $\lfloor t_1/K \rfloor = x_1$ and $\lfloor t_2/K \rfloor = x_2$.

Assume, by contradiction, that $d_\alpha(x_1, x_2) > 1$. By definition, this means that $\min\big((x_1 - x_2) \mod \alpha, (x_2 - x_1) \mod \alpha\big) > 1$. This implies that both $(x_1 - x_2) \mod \alpha > 1$ and $(x_2 - x_1) \mod \alpha > 1$. As $d_\beta(t_1, t_2) \leq d$, $\min\big((t_1 - t_2) \mod \beta, (t_2 - t_1) \mod \beta\big) \leq d$. Without loss of generality, assume that $(t_1 - t_2) \mod \beta \leq d$. There are two cases:

1. If $t_1 \geq t_2$, then $(t_1 - t_2) \mod \beta = t_1 - t_2$. So, $t_1 - t_2 \leq d$.

   Now, as $t_1 \geq t_2$, $x_1 \geq x_2$ by Remark 5.7. Hence $x_1 - x_2 = (x_1 - x_2) \mod \alpha > 1$. As x_1 and x_2 are natural numbers, this implies that $x_1 - x_2 \geq 2$. We rewrite the inequality as $x_1 K + \xi_1 - x_2 K - \xi_2 \geq 2K + \xi_1 - \xi_2$. Since $\xi_1, \xi_2 \in \{0, \ldots, K - 1\}$, we have $-K < \xi_1 - \xi_2 < K$ and therefore $x_1 K + \xi_1 - x_2 K - \xi_2 > K > d$. Hence, $t_1 - t_2 > d$, a contradiction.

2. If $t_1 < t_2$, then $(t_1 - t_2) \mod \beta = \beta + t_1 - t_2$. So, $\beta + t_1 - t_2 \leq d$.

   Now, as $t_1 < t_2$, $x_1 \leq x_2$ by Lemma 5.7. Hence $(x_1 - x_2) \mod \alpha = \alpha + x_1 - x_2 > 1$. As x_1 and x_2 are natural numbers, this implies that $\alpha + x_1 - x_2 \geq 2$. We rewrite the inequality as $\beta + x_1 K + \xi_1 - x_2 K - \xi_2 \geq 2K + \xi_1 - \xi_2$. Since $\xi_1, \xi_2 \in \{0, \ldots, K - 1\}$, we have $-K < \xi_1 - \xi_2 < K$ and therefore $\beta + x_1 K + \xi_1 - x_2 K - \xi_2 > K > d$. Hence, $\beta + t_1 - t_2 > d$, a contradiction.

■

As previous remarks, Lemma 5.14 will be used with the internal clock $t = c\frac{\beta}{\alpha} + \xi$: it expresses that once internal clocks have stabilized at a delay smaller than d, external clocks are at delay smaller than 1. We now prove that Algorithm 11 achieves the safety and liveness properties of SP_{SU} in any execution starting from a legitimate configuration.

Remark 5.8 (*Safety for $\alpha = 2$*)

Assume $\alpha = 2$. Every execution $e \in \mathcal{E}^0_{\mathcal{SU}}(\mathscr{L}_{\text{SU}})$ satisfies the safety of SP_{SU}. Indeed, there is only two possible values of clock, so there is at most two (consecutive) values of clock in the network.

Lemma 5.15 (*Safety for $\alpha = 3$*)

Assume $\alpha = 3$. Every execution $e \in \mathcal{E}^0_{\mathcal{SU}}(\mathscr{L}_{\text{SU}})$ satisfies the safety of SP_{SU}.

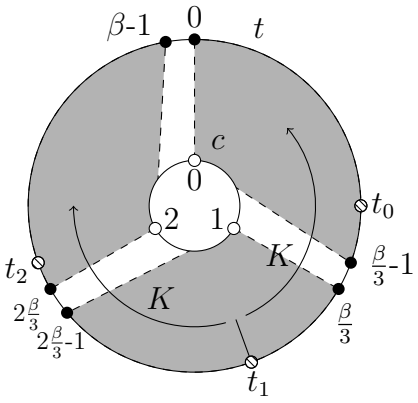


Figure 5.5 – Relative positions of t_0, t_1, and t_2.

Proof: If the number of nodes in the network is smaller than 3, trivially there is no more than two different values for clock c. Otherwise, let $\gamma \in \mathscr{L}_{\text{SU}}$ be a legitimate configuration w.r.t. SP_{SU} under $\mathcal{SU}$. Assume, by contradiction, that there are more than two different values of variable c in γ: $\exists p_0, p_1, p_2 \in V$ such that $\gamma(p_0).c = 0$, $\gamma(p_1).c = 1$ and $\gamma(p_2).c = 2$. We denote $t_j = \gamma(p_j).t$ for all $j \in \{0, 1, 2\}$ and using Remark 5.6, there exists $\xi_0, \xi_1, \xi_2 \in \{0, ..., \beta/3 - 1\}$ such that $t_0 = \xi_0$, $t_1 = \beta/3 + \xi_1$ and $t_2 = 2\beta/3 + \xi_2$ (see Figure 5.5).

As γ is legitimate w.r.t. SP_{SU}, the delay (d_β) between any two internal clocks c is upper bounded by $n-1$ (Remark 5.2) and using the assumption also upper bounded by $K = \beta/3$ (strict upper bound). So in particular

$$\begin{aligned} d_\beta(t_0, t_1) &= \beta/3 + \xi_1 - \xi_0 < n < K = \beta/3 \\ d_\beta(t_1, t_2) &= \beta/3 + \xi_2 - \xi_1 < n < K = \beta/3 \\ d_\beta(t_2, t_0) &= \beta/3 + \xi_0 - \xi_2 < n < K = \beta/3 \end{aligned}$$

It comes that $\xi_1 < \xi_0 < \xi_2 < \xi_1$, a contradiction.

Finally, as the set $\mathscr{L}_{\text{SU}}$ is closed (Lemma 5.12), we are done. ■

Lemma 5.16 (*Safety for $\alpha > 3$*)

Assumes $\alpha > 3$. Every execution $e \in \mathcal{E}^0_{\mathcal{SU}}(\mathscr{L}_{\text{SU}})$ satisfies the safety of SP_{SU}.

Proof: Let $\gamma \in \mathscr{L}_{\text{SU}}$: the delay ($\beta$) between any two internal clocks t in γ is upper bounded by $n-1$ and for any process, $p \in V$, $\gamma(p).c = \left\lfloor \frac{\alpha}{\beta}\gamma(p).t \right\rfloor$. Hence, using Lemma 5.14 with $d = n - 1 < K$, we have $\forall p, q \in V$, $d_\alpha(\gamma(p).c, \gamma(q).c) \leq 1$. As $\alpha > 3$, this proves that the variables c in γ have at most two different consecutive values.

Finally, as the set $\mathscr{L}_{\text{SU}}$ is closed (Lemma 5.12), we are done. ■

Lemma 5.17 (*Liveness*)

Every execution $e \in \mathcal{E}^0_{\mathcal{SU}}(\mathscr{L}_{\text{SU}})$ satisfies the liveness of SP_{SU}.

Proof: Let $e = (\gamma_i)_{i\geq 0} \in \mathcal{E}^0_{\mathcal{SU}}(\mathscr{L}_{\text{SU}})$. Let p be a process. γ_0 is a legitimate configuration of $\mathcal{WU}$ so p increments infinitely often $p.t$ using $\mathcal{SU}$-**N**-action (by Theorem 5.5 and Remark 5.4). So $p.t$ goes through each integer value between 0 and $\beta - 1$ infinitely often (in increasing order). Hence, by Remark 5.6, $p.c$ is incremented infinitely often and goes through each integer value between 0 and $\alpha - 1$ (in increasing order). ■

Proof of Theorem 5.8: Lemma 5.12 (closure), Lemma 5.13 (convergence), Lemmas 5.16-5.17 and Remark 5.8 (correctness) prove that Algorithm $\mathcal{SU}$ is self-stabilizing for SP_{SU} in any arbitrary connected anonymous network assuming a distributed unfair daemon. ■

Complexity Analysis. We now give some complexity results about Algorithm $\mathcal{SU}$. Precisely, a bound on the stabilization time of $\mathcal{SU}$ is given in Theorem 5.9. Then, a delay between any two consecutive clocks increments, which holds once $\mathcal{SU}$ has stabilized, is given in Theorem 5.10.

Theorem 5.9

The stabilization time of $\mathcal{SU}$ to $\mathscr{L}_{\text{SU}}$ is at most $n + (\mu + 1)\mathcal{D} + 1$ rounds, where n (resp. $\mathcal{D}$) is the size (resp. diameter) of the network and μ is a parameter satisfying $\mu \geq \max(n, 2)$.

Proof: Let $(\gamma_i)_{i\geq 0} \in \mathcal{E}^0_{\mathcal{SU}}$. The behavior of the t-variables in $\mathcal{SU}$ is similar to that of $\mathcal{WU}$ (Remark 5.4), which stabilizes in at most $n + \mu\mathcal{D}$ rounds (see Theorems 5.6 and 5.7) to weak unison. So, in $n + \mu\mathcal{D}$ rounds, the delay between the t-clocks of any two arbitrary far processes is at most $n - 1$ (Remark 5.2). If c-variables are well-calculated according to t-variables, *i.e.*if $c = \left\lfloor \frac{\alpha}{\beta} t \right\rfloor$, then the delay between the c-clocks of any two arbitrary far processes is at most 1 (Lemma 5.14). In at most $\mathcal{D} + 1$ additional rounds, each process executes $\mathcal{SU}$-**N**-action (Lemma 5.11) and updates its c-variable according to its t-variable. Hence, in at most $n + (\mu + 1)\mathcal{D} + 1$ rounds, the system reaches a legitimate configuration. ■

Theorem 5.10

After convergence of $\mathcal{SU}$ to $\mathscr{L}_{\text{SU}}$, each process p increments its clock $p.c$ at least once every $\mathcal{D} + \frac{\beta}{\alpha}$ rounds, where $\mathcal{D}$ is the diameter of the network.

Proof: If $\mathcal{SU}$ converged to $\mathscr{L}_{\text{SU}}$, by Remark 5.4 and Lemma 5.11, after $\mathcal{D} + \frac{\beta}{\alpha}$ rounds, p increments $p.t$ at least $\frac{\beta}{\alpha}$ times. Now, by Remark 5.6, if a t-variable is incremented $\frac{\beta}{\alpha}$ times, then its corresponding c-variable is incremented once. ■

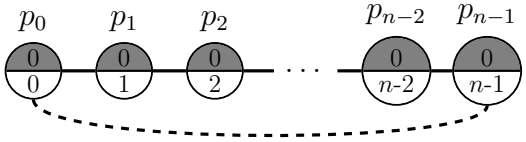


Figure 5.6 – Link addition.

5.6 Gradually Stabilizing Strong Unison

We now propose Algorithm $\mathcal{DSU}$ (Algorithm 12), a gradually stabilizing variant of Algorithm $\mathcal{SU}$. First, to maintain a finite period for internal clocks, we need to assume that the number of processes in any reachable configuration never exceeds some bound $N \geq n$. Indeed, in compliance with Algorithm $\mathcal{SU}$, the parameter μ in Algorithm $\mathcal{DSU}$ should be fixed to a value greater than or equal to the maximum between 2 and N. Then, according to the results shown in Section 5.4 (Theorems 5.1-5.2 and Corollary 5.1), we consider the following dynamic pattern:

$$\mathsf{BULCC}(G_i, G_j) \equiv |V_j| \leq N \vee (\alpha > 3 \Rightarrow \mathsf{ULC}(G_i, G_j)) \vee \mathsf{C}(G_i, G_j)$$

BULCC stands for *Bounded number of nodes, Under Local Control, and Connected*. Precisely, after any BULCC-dynamic step (such a step may include several topological events) from a configuration of $\mathscr{L}_{\text{SU}}$ (the set of legitimate configurations w.r.t. strong unison) $\mathcal{DSU}$ maintains (external) clocks almost synchronized during the convergence to strong unison, since it immediately satisfies partial weak unison, then converges in at most one round to weak unison, and finally re-stabilizes to strong unison.

In the following, we present in Subsection 5.6.1 the general principles of the convergence of $\mathcal{DSU}$ after a BULCC-dynamic step occurs from a configuration of $\mathscr{L}_{\text{SU}}$. Then, we show the gradual stabilization of $\mathcal{DSU}$ in Subsection 5.6.2.

5.6.1 Overview of Algorithm $\mathcal{DSU}$

We now explain step by step how we modify Algorithm $\mathcal{SU}$to obtain our gradually stabilizing algorithm, $\mathcal{DSU}$. We consider any BULCC-dynamic step $\gamma_i \mapsto^{d,\mathsf{BULCC}}_{\mathcal{DSU}} \gamma_{i+1}$ such that $\gamma_i \in \mathscr{L}_{\text{SU}}$, *i.e.*, the set of legitimate configurations w.r.t. strong unison. Since, $\mathscr{L}_{\text{SU}}$ is closed (Lemma 5.12), the set of configurations reachable from $\mathscr{L}_{\text{SU}}$ after one BULCC-dynamic step (which may also include process activations) is the same as the one reachable from $\mathscr{L}_{\text{SU}}$ after BULCC-dynamic step made of topological events only (see Theorem 5.13). At the light of this result, we consider, without the loss of generality, no process moves during $\gamma_i \mapsto^{d,\mathsf{BULCC}}_{\mathcal{DSU}} \gamma_{i+1}$.

Assume first that $\gamma_i \mapsto^{d,\mathsf{BULCC}}_{\mathcal{DSU}} \gamma_{i+1}$ contains link additions only. Adding a link (see the dashed link in Figure 5.6) can break the safety of weak unison on internal clocks. Indeed, it may create a delay greater than one between two new neighboring t-clocks. Nevertheless, the delay between any two t-clocks remains bounded by $n-1$ (recall that n is the number of processes initially in the network), consequently, no process will reset its t-clock (Figure 5.6 shows a worst case). Moreover, c-clocks still satisfy strong unison immediately after the link addition. Besides, since increments are constrained

Algorithm 12 – Actions of Process p in Algorithm $\mathcal{DSU}$.

Inputs.

- $\alpha \in \mathbb{N}$ such that $\alpha \geq 2$
- $\mu \in \mathbb{N}$ such that $\mu \geq \max(N, 2)$
- $\beta \in \mathbb{N}$ such that $\beta > \mu^2$ and $\exists K$ such that $K > \mu$ and $\beta = K\alpha$

Variables.

- $p.c \in \{0, \ldots, \alpha - 1\} \cup \{\bot\}$
- $p.t \in \{0, \ldots, \beta - 1\}$

Predicates.

$$Locked(p) \quad \equiv \quad p.c = \bot \vee \exists q \in p.\mathcal{N}, q.c = \bot$$

Guards.

$$\begin{aligned} NormalStep(p) &\equiv \neg Locked(p) \wedge \forall q \in p.\mathcal{N}, p.t \preceq_{\beta,\mu} q.t \\ ResetStep(p) &\equiv \neg Locked(p) \wedge \big(\exists q \in p.\mathcal{N}, d_\beta(p.t, q.t) > \mu \wedge p.t \neq 0\big) \\ JoinStep(p) &\equiv p.c = \bot \end{aligned}$$

Actions.

$$\begin{array}{llll} \mathcal{DSU}\textbf{-N} :: NormalStep(p) \rightarrow & p.t := (p.t+1) \mod \beta \\ & p.c := \left\lfloor \frac{\alpha}{\beta} p.t \right\rfloor \\ \mathcal{DSU}\textbf{-R} :: ResetStep(p) \rightarrow & p.t := 0 \\ & p.c := 0 \\ \mathcal{DSU}\textbf{-J} :: JoinStep(p) \rightarrow & p.t := MinTime(p) \\ & p.c := \left\lfloor \frac{\alpha}{\beta} p.t \right\rfloor \\ \text{bootstrap} :: join(p) \rightarrow & p.t := 0 \\ & p.c := \bot \end{array}$$

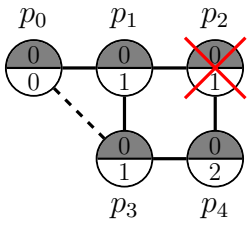
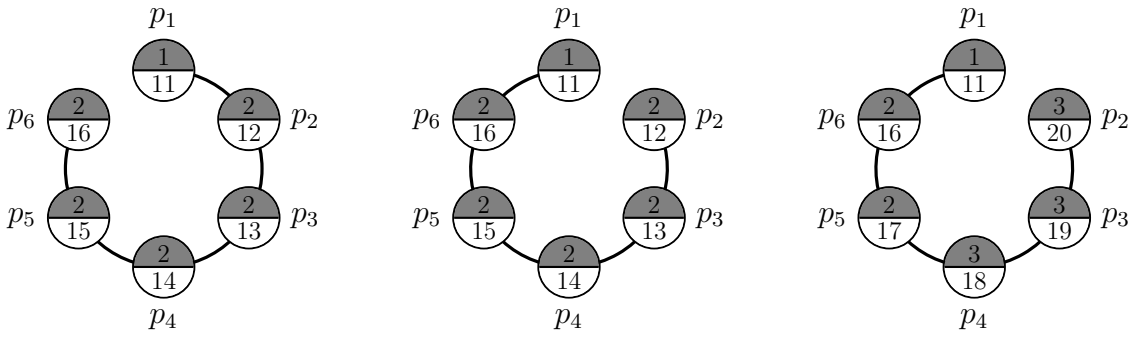


Figure 5.7 – Removals.

by neighboring clocks, adding links only reinforces those constraints. Thus, the delay between internal clocks of arbitrary far processes remains bounded by $n-1$, and so strong unison remains satisfied, in all subsequent static steps. Consider again the example in Figure 5.6: before $\gamma_i \mapsto^{d,\mathsf{BULCC}}_{\mathcal{DSU}} \gamma_{i+1}$, p_{n-1} had only to wait until p_{n-2} increments tp_{n-2} in order to be able to increment its own t-clock; yet after the step, it also has to wait for p_0 until its internal clock reaches at least $n-1$.

Assume now that $\gamma_i \mapsto^{d,\mathsf{BULCC}}_{\mathcal{DSU}} \gamma_{i+1}$ contains process and link removals only. By definition of BULCC, the network remains connected. Hence, constraints between (still existing) neighbors are maintained: the delay between t-clocks of two neighbors remains bounded by one, see the example in Figure 5.7: process p_2 and link $\{p_0, p_3\}$ are removed. So, weak unison on t-clocks remains satisfied and so is strong unison on c-clocks.



(a) Initial configuration satisfying strong unison.

(b) After one dynamic step: link $\{p_1, p_2\}$ disappears and link $\{p_1, p_6\}$ is created.

(c) Some steps later, strong unison is violated.

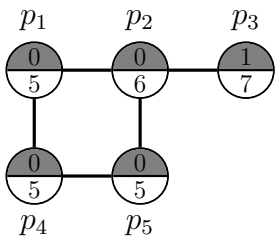
Figure 5.8 – Example of execution where one link is added and another is removed: $\mu = 6$, $\alpha = 7$, and $\beta = 42$.

Consider now a more complex scenario, where $\gamma_i \mapsto^{d,\mathsf{BULCC}}_{\mathcal{DSU}} \gamma_{i+1}$ contains link additions as well as process and/or link removals. Figure 5.8 shows an example of such a scenario, where safety of strong unison is violated. As above, the addition of link $\{p_1, p_6\}$ in Figure 5.8(b) leads to a delay between t-clocks of these two (new) neighbors which is greater than one (here 5). However, the removal of link $\{p_1, p_2\}$, also in Figure 5.8(b), relaxes the neighborhood constraint on p_2: p_2 can now increment without waiting for p_1. Consequently, executing Algorithm $\mathcal{SU}$ does not ensure that the delay between t-clocks of any two arbitrary far processes remains bounded by $n-1$, *e.g.*, after several static steps from Figure 5.8(b), the system can reach Figure 5.8(c), where the delay between p_1 and p_2 is 9 while $n - 1 = 5$. Since c-clock values are computed from t-clock values, we also cannot guarantee that there is at most two consecutive c-clock values in the system, *e.g.*, in Figure 5.8(c) we have: $p_1.c = 1$, $p_6.c = 2$, and $p_2.c = 3$. Again, in the worst case scenario, after $\gamma_i \mapsto^{d,\mathsf{BULCC}}_{\mathcal{DSU}} \gamma_{i+1}$, the delay between two neighboring t-clocks is bounded by $n - 1$. Moreover, t-clocks being computed like in Algorithm $\mathcal{WU}$, we can use two of its useful properties (see [Bou07]):

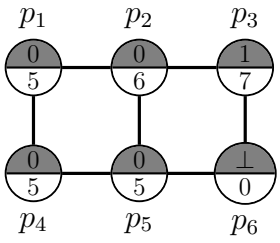
1. when the delay between every pair of neighboring t-clocks is at most μ with $\mu \geq n$, the delay between these clocks remains bounded by μ because processes never reset;
2. furthermore, from such configurations, the system converges to a configuration from which the delay between the t-clocks of every two neighbors is at most one.

So, keeping $\mu \geq n$, processes will not reset after that BULCC-dynamic step and the delay between any two neighboring t-clocks will monotonically decrease from at most $n - 1$ to at most one. Consequently, the delay between any two neighboring c-clocks (which are computed from t-clocks) will stay at most one, *i.e.*, weak unison will be satisfied all along the convergence to strong unison.

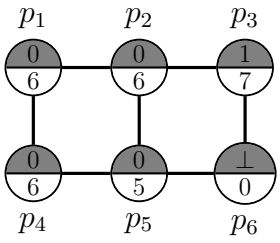
Assume now that a process p joins the system during $\gamma_i \mapsto^{d,\mathsf{BULCC}}_{\mathcal{DSU}} \gamma_{i+1}$. The event $join_p$ occurs and triggers the new specific action *bootstrap* that initialized p to its bootstate: p sets $p.c$ to a specific *bootstate* value, noted $\perp$ (meaning that its output is currently



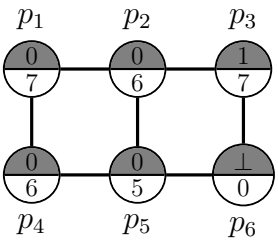
(a) Initial configuration satisfying strong unison.



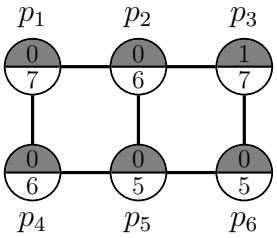
(b) After the dynamic step where process p_6 joins, p_3 and p_5 are locked. p_1 and p_4 are enabled to execute $\mathcal{DSU}$-**N**-action.



(c) p_4 is disabled. $\mathcal{DSU}$-**N**-action is enabled at p_1.



(d) Now, p_1 is disabled because of p_2 and p_4. p_6 is the only enabled process.



(e) p_6 executes $\mathcal{DSU}$-**J**-action and initializes its clocks.

Figure 5.9 – Example of execution where the daemon delays the first step of a new process: $\mu = 6$, $\alpha = 6$, and $\beta = 42$.

undefined), and $p.t$ to 0. By definition and from the previous discussion, the system immediately satisfies partial unison since it only depends on processes that were in the system before the BULCC-dynamic step. Now, to ensure that weak unison is satisfied within a round, we add the $\mathcal{DSU}$-**J**-action which is enabled as soon as the process is in bootstate. This action initializes the two clocks of p according to the minimum t-clock value of its neighbors that are not in bootstate, if any. To that goal, we use the function $MinTime(p)$ given below.

$$MinTime(p) = \begin{cases} 0 & \textbf{if } \forall q \in p.\mathcal{N}, q.clock = \perp, \\ \min\{q.t : q \in p.\mathcal{N} \wedge q.clock \neq \perp\} & \textbf{otherwise.} \end{cases}$$

The value of $p.c$ is then computed according to the value of $p.t$. Remark that $MinTime(p)$ returns 0 when p and all its neighbors have their respective c-clock equal to $\perp$: if the BULCC-dynamic step replaces all nodes by new ones, then the system reaches in a configuration where all c-clocks are equal to $\perp$, and $\mathcal{DSU}$ still ensures gradual stabilization in this case.

Then, to prevent the unfair daemon from blocking the convergence to a configuration containing no $\perp$-values, we should also forbid processes with non-$\perp$ c-clock values to increment while there are c-clocks with $\perp$-values in their neighborhood. So, we define the

predicate *Locked* which holds for a given process p when either $p.c = \perp$, or at least one of its neighbors q satisfies $q.clock = \perp$. We then enforce the guard of both normal and reset actions, so that no *Locked* process can execute them. See $\mathcal{DSU}$-**N**- and $\mathcal{DSU}$-**R**-actions. This ensures that t-clocks are initialized first by $\mathcal{DSU}$-**J**-action, before any value in their neighborhood increments.

Finally, notice that all the previous explanation relies on the fact that, once the system recovers from process additions (*i.e.*, once no $\perp$ value remains), the algorithm behaves exactly the same as Algorithm $\mathcal{SU}$. Hence, it has to match the assumptions made for $\mathcal{SU}$, in particular, the ones on α and β. However the constraint on μ has to be adapted, since μ should be greater than or equal to the actual number of processes in the network and this number may increase. Now, this number is assumed to be bounded by N. So, we now require that $\mu \geq \max(N, 2)$.

We now consider the example execution of Algorithm $\mathcal{DSU}$ in Figure 5.9. This execution starts in a configuration legitimate *w.r.t.* the strong unison, see Figure 5.9(a). Then, one BULCC-dynamic step happens (step (a)$\mapsto$(b)), where a process p_6 joins the system. We now try to delay as long as possible the execution of $\mathcal{DSU}$-**J**-action by p_6. In configuration (b), p_3 and p_5, the new neighbors of p_6, are locked. They will remain disabled until p_6 executes $\mathcal{DSU}$-**J**-action. p_1 and p_4 execute $\mathcal{DSU}$-**N**-action in (b)$\mapsto$(c). Then, p_4 is disabled because of p_5 and p_1 executes $\mathcal{DSU}$-**N**-action in (c)$\mapsto$(d). In configuration (d), p_1 is from now on disabled: p_1 must wait until p_2 and p_4 get t-clock value 7. p_6 is the only enabled process, so the unfair daemon has no other choice but selecting p_6 to execute $\mathcal{DSU}$-**J**-action in the next step.

5.6.2 Correctness of $\mathcal{DSU}$

Self-stabilization w.r.t. SP_{SU}.

Remark 5.9

By definition, $N \geq n$, so, whenever the parameters α, μ, and β satisfy the constraints of Algorithm $\mathcal{DSU}$, they also satisfy all constraints of Algorithm $\mathcal{SU}$.

Remark 5.10

In $\mathcal{DSU}$, if all c-variables have values different from $\perp$, predicates JoinStep and Locked are FALSE. *Furthermore, no action can assign $\perp$ to c during a static step. Consequently, when all c-variables have values different from $\perp$, and as far as no topological change occurs, Algorithms $\mathcal{DSU}$ and $\mathcal{SU}$ are syntactically identical. So, fixing the initial graph and the three parameters α, μ, and β, we obtain that the set of executions $\mathcal{E}^0_{\mathcal{SU}}$ and the set of executions $\mathcal{E}^0_{\mathcal{DSU}}(NoBot)$ are identical, where*

$$NoBot = \{\gamma \in \mathcal{C} \ : \ \forall p \in V, \gamma(p.c) \neq \perp\}$$

By Remarks 5.10 and 5.9, results of Algorithm $\mathcal{SU}$ about $\mathscr{L}_{\text{SU}}$ (see Definition 5.9 page 150) also hold for Algorithm $\mathcal{DSU}$. Hence, follows.

Lemma 5.18 (*Closure and Correctness of $\mathscr{L}_{\text{SU}}$ under $\mathcal{DSU}$*)

$\mathscr{L}_{\text{SU}}$ is closed under $\mathcal{DSU}$, and for every execution $e \in \mathcal{E}^0_{\mathcal{DSU}}(\mathscr{L}_{\text{SU}})$, $SP_{\text{SU}}(e)$.

Lemma 5.19

For any execution $(\gamma_i)_{i\geq 0} \in \mathcal{E}^0_{\mathcal{DSU}}$, $\exists j \geq 0$ such that $\forall k \geq j$, $\forall p \in V$, $\gamma_k(p).c \neq \perp$.

Proof: Let $e = (\gamma_i)_{i\geq 0} \in \mathcal{E}^0_{\mathcal{DSU}}$. For any $i \geq 0$, we note $Bottom(\gamma_i) = \{p \in V : \gamma_i(p).c = \perp\}$. As $\mathcal{DSU}$-**N**, $\mathcal{DSU}$-**R**, and $\mathcal{DSU}$-**J**-action do not create any $\perp$-value, $\forall i > 0$, $Bottom(\gamma_i) \subseteq Bottom(\gamma_{i-1})$. Now, assume by contradiction that $\exists p \in V$ such that $\forall i \geq 0$, $p \in Bottom(\gamma_i)$. Since, the number of nodes is constant, there is a configuration γ_s, $s \geq 0$, from which no $\perp$-value disappears anymore, *i.e.*$\forall p \in V$, $p \in Bottom(\gamma_s) \Rightarrow \forall i \geq s, p \in Bottom(\gamma_i)$.

If $Bottom(\gamma_s) = V$, every process is enabled for $\mathcal{DSU}$-**J**-action. So, the unfair daemon selects at least one process to execute $\mathcal{DSU}$-**J** and sets its c-variable to a value different from $\perp$, a contradiction with the definition of γ_s.

Hence there is at least one process that is not in $Bottom(\gamma_s)$. Again, if the only enabled processes are in $Bottom(\gamma_s)$, then the unfair daemon has no other choice but selecting one of them, a contradiction. So, $\forall i \geq s$, there exists a process that is enabled in γ_i but which is not in $Bottom(\gamma_i)$. Remark that this implies in particular that e is an infinite execution.

Now, let consider the subgraph $\mathcal{G}'$ of $\mathcal{G}$ induced by $V \backslash Bottom(\gamma_s)$. $\mathcal{G}'$ is composed of a finite number of connected components and, as e is infinite, there is an infinite number of actions of e executed in (at least) one of these components. Let $\mathcal{G}'' = (V'', E'')$ be such a connected component.

Let $e' = (\gamma'_i)_{i\geq 0}$ be the projection of e on $\mathcal{G}''$ and t-variable: $\forall i \geq 0$, $\forall x \in V''$, $\gamma'_i(x).t = \gamma_i(x).t$. We construct $e'' = (\gamma''_j)_{j\geq 0}$ from e' by removing duplicate configurations with the following inductive schema:

- $\gamma''_0 = \gamma'_0$,
- and, $\forall j > 0$, if $\gamma''_0 \dots \gamma''_j$ represents $\gamma'_0 \dots \gamma'_k$ without duplicate configurations, $\gamma''_{j+1} = \gamma'_{next}$, where $next = \min\{l > k : \gamma'_l \neq \gamma'_k\}$. (Notice that $next$ is always defined as there is an infinite number of actions executed in $\mathcal{G}''$.)

Let $L = \{p \in V'' : \exists q \in Bottom(\gamma_s), \{p, q\} \in E\}$ be the set of processes that are neighbors of a $Bottom(\gamma_s)$ process in $\mathcal{G}$. As $\mathcal{G}$ is connected, L is not empty. Furthermore, during the execution e, *Locked* holds forever for processes in L, hence are disabled. As a consequence, in execution e'', no process in L can execute a static step. Now, from Remark 5.4 and 5.10, and since γ''_0 contains no $\perp$-value, e'' is also an execution of $\mathcal{WU}$ in graph $\mathcal{G}''$. The fact that existing processes (from the non-empty set L) never increment their clocks during an infinite execution e'' of $\mathcal{WU}$ is a contradiction with the liveness of the weak unison (Specification 5.1), Remark 5.9, and Theorem 5.5 which states that $\mathcal{WU}$ is self-stabilizing for weak unison under an unfair daemon. ■

Lemma 5.20 (*Convergence to $\mathscr{L}_{\text{SU}}$*)

$\mathcal{C}$ (the set of all possible configurations) converges under $\mathcal{DSU}$ to the set of legitimate configurations $\mathscr{L}_{\text{SU}}$.

Proof: Let $(\gamma_i)_{i\geq 0} \in \mathcal{E}^0_{\mathcal{DSU}}$. Using Lemma 5.19, $\exists j \geq 0$ such that $\forall k \geq j$, $\forall p \in V$, $\gamma_k(p).c \neq \perp$. After γ_j, the execution of the system, $(\gamma_k)_{k\geq j}$, is also a possible execution of $\mathcal{SU}$ (by Remark 5.10). Hence, it converges to a configuration γ_k $(k \geq j)$ in $\mathscr{L}_{\text{SU}}$, by Remark 5.9 and Lemma 5.13. ■

Using Lemmas 5.18 and 5.20, we can deduce the following theorem.

Theorem 5.11 (*Self-stabilization of $\mathcal{DSU}$* w.r.t. *strong unison*)

Algorithm $\mathcal{DSU}$ is self-stabilizing for SP_{SU} in any arbitrary connected anonymous network assuming a distributed unfair daemon.

Theorem 5.12 states the stabilization time of $\mathcal{DSU}$.

Theorem 5.12

The stabilization time of $\mathcal{DSU}$ to $\mathscr{L}_{\text{SU}}$ is at most $n + (\mu + 1)\mathcal{D} + 2$, where n (resp. $\mathcal{D}$) is the size (resp. diameter) of the network, and μ is a parameter satisfying $\mu \geq \max(2, N)$.

Proof: Let $(\gamma_i)_{i\geq 0} \in \mathcal{E}^0_{\mathcal{DSU}}$. If there are some processes p such that $\gamma_0(p).c = \perp$, $\mathcal{DSU}$-**J**-action is continuously enabled at p. So, after at most one round $p.c \neq \perp$, for every process p. Afterwards, the behavior of the algorithm is identical to the one of $\mathcal{SU}$ (Remarks 5.9 and 5.10), which stabilizes in at most $n+(\mu+1)\mathcal{D}+1$ rounds (see Theorem 5.9). Hence, in at most $n + (\mu + 1)\mathcal{D} + 2$ rounds, the system reaches a legitimate configuration. ■

Immediate Stabilization to SP_{PU} after one BULCC-Dynamic Step from $\mathscr{L}_{\text{SU}}$.

Definition 5.10 (*Legitimate Configurations of $\mathcal{DSU}$* w.r.t. *SP_{PU}*)

A configuration γ_i of $\mathcal{DSU}$ is *legitimate* w.r.t. SP_{PU} if and only if

1. $\forall p \in V_i$, $\gamma_i(p).c \neq \perp \Rightarrow \gamma_i(p).c = \left\lfloor \frac{\alpha}{\beta} t\gamma_i(p) \right\rfloor$; and
2. if $\alpha > 3$, then the following three additional conditions hold:
   a) $\big[\forall p \in V_i, \big(\gamma_i(p).c = \perp \Rightarrow (\forall q \in \gamma_i(p).\mathcal{N}, \gamma_i(q).c \in \{0, \perp\})\big)\big]$
   $\vee \big[\forall p \in V_i, \big(\gamma_i(p).c = \perp \Rightarrow (\exists q \in \gamma_i(p).\mathcal{N}, \gamma_i(q).c \neq \perp)\big)\big]$
   b) $\forall p \in V_i, \forall q \in \gamma_i(p).\mathcal{N}$,
   $$\gamma_i(p).c \neq \perp \wedge \gamma_i(q).c \neq \perp \Rightarrow d_\beta(\gamma_i(p).t, \gamma_i(q).t) \leq \mu;$$
   c) $\forall p, q \in V_i$,
   $$\gamma_i(p).c \neq \perp \wedge (\exists x \in \gamma_i(p).\mathcal{N}, \gamma_i(x).c = \perp) \wedge$$
   $$\gamma_i(q).c \neq \perp \wedge (\exists y \in \gamma_i(q).\mathcal{N}, \gamma_i(y).c = \perp) \Rightarrow$$
   $$d_\beta(\gamma_i(p).t, \gamma_i(q).t) \leq \mu.$$

We denote by $\mathscr{L}_{\text{PU}}$ the set of legitimate configurations of $\mathcal{DSU}$ w.r.t. SP_{PU}.

Lemma 5.21 (*Closure of $\mathscr{L}_{\text{PU}}$ under $\mathcal{DSU}$*)

$\mathscr{L}_{\text{PU}}$ is closed under $\mathcal{DSU}$.

Proof: Let $\gamma_i \mapsto^s \gamma_{i+1}$ be a static step of $\mathcal{DSU}$ such that $\gamma_i \in \mathscr{L}_{\text{PU}}$. By definition, $\mathcal{DSU}$-**R** is disabled in γ_i, for all processes: a process can only execute $\mathcal{DSU}$-**N** or $\mathcal{DSU}$-**J**-action depending whether its c-clock is $\perp$ or not.

Point 1 of Definition 5.10. Let $p \in V_{i+1}$ such that $\gamma_{i+1}(p).c \neq \perp$. Two cases are possible: either p executes no action during $\gamma_i \mapsto^s \gamma_{i+1}$ and the constraint between $p.t$ and $p.c$ has been preserved, or p executes $\mathcal{DSU}$-**J** or $\mathcal{DSU}$-**N**-action. In the latter case, the assignment of the action ensures the constraint.

For the three next points, we assume that $\alpha > 3$ (otherwise, those constraints are trivially preserved).

Point 2a of Definition 5.10. Since $\gamma_i \in \mathscr{L}_{\text{PU}}$ and by Definition 5.10.2a, two cases are possible.

Assume $\forall p \in V_i, (\gamma_i(p).c = \perp \Rightarrow (\forall q \in \gamma_i(p).\mathcal{N}, \gamma_i(q).c \in \{0, \perp\}))$. Neither $\mathcal{DSU}$-**N** nor $\mathcal{DSU}$-**J**-action sets c to $\perp$. Then, let $p \in V_i$ such that $\gamma_i(p).c = \perp$. Let q be a neighbor of p in γ_i. If $\gamma_i(q).c = 0$, then $\gamma_{i+1}(q).c = 0$, since q is disabled ($Locked(q)$ holds in γ_i because of p). If $\gamma_i(q).c = \perp$, then $\gamma_{i+1}(q).c \in \{0, \perp\}$ depending on whether or not q executes $\mathcal{DSU}$-**J**, since the c-clock values of all its neighbors are 0 or $\perp$, by hypothesis, and since the c-clock values of its non-$\perp$ neighbors are well computed according to their t-clock values, by Definition 5.10.1. Hence, the constraint is preserved in this case, and we are done.

Otherwise, $\forall p \in V_i, (\gamma_i(p).c = \perp \Rightarrow (\exists q \in \gamma_i(p).\mathcal{N}, \gamma_i(q).c \neq \perp))$. Since neither $\mathcal{DSU}$-**N** nor $\mathcal{DSU}$-**J**-action sets $p.c$ to $\perp$, this constraint is also preserved in this case.

Point 2b of Definition 5.10. Let p, q be two neighbors such that $\gamma_{i+1}(p).c \neq \perp$ and $\gamma_{i+1}(q).c \neq \perp$.

1. Assume that $\gamma_i(p).c \neq \perp$ and $\gamma_i(q).c \neq \perp$. As $\gamma_i \in \mathscr{L}_{\text{PU}}$, we have $d_\beta(\gamma_i(p).t, \gamma_i(q).t) \leq \mu$, by Definition 5.10.2b. So, p and q can only execute $\mathcal{DSU}$-**N**-action during $\gamma_i \mapsto^s \gamma_{i+1}$. If both p and q, or none of them, execute $\mathcal{DSU}$-**N**-action, the delay between $p.t$ and $q.t$ remains the same. If only one of them, say p, executes $\mathcal{DSU}$-**N**-action, $\gamma_i(p).t \preceq_{\beta,\mu} \gamma_i(q).t$. So, either $\gamma_i(p).t = \gamma_i(q).t$ and $d_\beta(\gamma_{i+1}(p).t, \gamma_{i+1}(q).t) = 1 \leq \mu$, or the increment of p.$t$decreases the delay between p.$t$and q.$t$and again we have $d_\beta(\gamma_{i+1}(p).t, \gamma_{i+1}(q).t) \leq \mu$.
2. Assume that $\gamma_i(p).c = \perp$ and $\gamma_i(q).c \neq \perp$. Let x be the neighbor of p in γ_i such that $\gamma_i(x).c \neq \perp$ with the minimum t-value (x is defined because $\gamma_i(q).c \neq \perp$). Since q and x are both neighbors of p in γ_i and $\gamma_i \in \mathscr{L}_{\text{PU}}$, $d_\beta(\gamma_i(x).t, \gamma_i(q).t) \leq \mu$, by Definition 5.10.2c. Moreover, q is disabled in γ_i because of p ($Locked(q)$ holds in γ_i), so $\gamma_{i+1}(q).t = \gamma_i(q).t$. Necessarily, p executes $\mathcal{DSU}$-**J**-action in $\gamma_i \mapsto^s \gamma_{i+1}$, since $\gamma_{i+1}(p).c \neq \perp$. Hence, $\gamma_{i+1}(p).t = \gamma_i(x).t$ and $d_\beta(\gamma_{i+1}(p).t, \gamma_{i+1}(q).t) \leq \mu$.
3. Assume that $\gamma_i(p).c \neq \perp$ and $\gamma_i(q).c = \perp$. This case is similar to the previous one.
4. Assume that $\gamma_i(p).c = \perp$ and $\gamma_i(q).c = \perp$. As $\gamma_{i+1}(p).c \neq \perp$ and $\gamma_{i+1}(q).c \neq \perp$, p and q necessarily move during $\gamma_i \mapsto^s \gamma_{i+1}$. Since $\gamma_i \in \mathscr{L}_{\text{PU}}$, two cases are possible, by Definition 5.10.2a.

If $\forall v \in V_i, (\gamma_i(v).c = \bot \Rightarrow (\forall w \in \gamma_i(v).\mathcal{N}, \gamma_i(w).c \in \{0, \bot\}))$, then $\gamma_{i+1}(p).c = \gamma_{i+1}(q).c = 0$ (owing the fact that the c-clock values of all their non-$\bot$ neighbors are well computed according to their t-clock values, by Definition 5.10.1), and we are done.
Otherwise, $\forall v \in V_i, (\gamma_i(v).c = \bot \Rightarrow (\exists w \in \gamma_i(v).\mathcal{N}, \gamma_i(w).c \neq \bot))$. So, in γ_i, we have: $\exists x \in p.\mathcal{N}$ such that $\gamma_i(x).c \neq \bot \wedge \gamma_i(x).t = MinTime(p)$ and $\exists y \in q.\mathcal{N}$ such that $\gamma_i(y).c \neq \bot \wedge \gamma_i(y).t = MinTime(q)$, by Definition 5.10.2a. Moreover, x and y have neighbors whose c-variables equal $\bot$ (p and q, respectively), we have $d_\beta(\gamma_i(x).t, \gamma_i(y).t) \leq \mu$, by Definition 5.10.2c. Since p and q execute $\mathcal{DSU}$-**J**-action, $\gamma_{i+1}(p).t = \gamma_i(x).t$ and $\gamma_{i+1}(q).t = \gamma_i(y).t$, and we are done.

Point 2c of Definition 5.10. Let p, q be two processes such that $\gamma_{i+1}(p).c \neq \bot$, $\exists x \in \gamma_{i+1}(p).\mathcal{N}$ with $\gamma_{i+1}(x).c = \bot$, $\gamma_{i+1}(q).c \neq \bot$, and $\exists y \in \gamma_{i+1}(q).\mathcal{N}$ with $\gamma_{i+1}(y).c = \bot$. As no enabled action can set variable c to $\bot$, $\gamma_i(x).c = \bot$ and $\gamma_i(y).c = \bot$.

1. Assume that $\gamma_i(p).c \neq \bot$ and $\gamma_i(q).c \neq \bot$. As $\gamma_i \in \mathscr{L}_{\text{PU}}$, $d_\beta(\gamma_i(p).t, \gamma_i(q).t) \leq \mu$, by Definition 5.10.2b. Now, p and q are disabled in γ_i because of x and y ($Locked(p)$ and $Locked(q)$ hold in γ_i). Hence, $d_\beta(\gamma_{i+1}(p).t, \gamma_{i+1}(q).t) \leq \mu$.
2. Assume that $\gamma_i(p).c = \bot$ and $\gamma_i(q).c \neq \bot$. Since $\gamma_i \in \mathscr{L}_{\text{PU}}$, two cases are possible, by Definition 5.10.2a.
   Assume $\forall v \in V_i, (\gamma_i(v).c = \bot \Rightarrow (\forall w \in \gamma_i(v).\mathcal{N}, \gamma_i(w).c \in \{0, \bot\}))$. Then, since $\gamma_i(y).c = \bot$, $\gamma_i(q).c = 0$. Then, $\gamma_{i+1}(p).c = \gamma_{i+1}(q).c = 0$ (p necessarily moves in $\gamma_i \mapsto^s \gamma_{i+1}$ and gets clock 0 owing the fact that the c-clock values of all its non-$\bot$ neighbors are well computed according to their t-clock values, by Definition 5.10.1; moreover q is disabled in γ_i since $Locked(q)$ hold because of y), and we are done.
   Otherwise, $\forall v \in V_i, (\gamma_i(v).c = \bot \Rightarrow (\exists w \in \gamma_i(v).\mathcal{N}, \gamma_i(w).c \neq \bot))$. Then, $\exists x' \in \gamma_i(p).\mathcal{N}$ such that $c\gamma_i(x') \neq \bot \wedge t\gamma_i(x') = MinTime(p)$ in γ_i. By Definition 5.10.2c, $d_\beta(t\gamma_i(x'), \gamma_i(q).t) \leq \mu$ because they have neighbors whose c-variables equal $\bot$ (p and y, respectively). Moreover, q is disabled in γ_i because of y: $\gamma_{i+1}(q).t = \gamma_i(q).t$. Finally, $\gamma_{i+1}(p).t = \gamma_i(x').t$ since p executes $\mathcal{DSU}$-**J**-action. So, $d_\beta(\gamma_{i+1}(p).t, \gamma_{i+1}(q).t) \leq \mu$.
3. Assume that $\gamma_i(p).c \neq \bot$ and $\gamma_i(q).c = \bot$. The case is similar to the previous one.
4. Assume that $\gamma_i(p).c = \bot$ and $\gamma_i(q).c = \bot$. Since $\gamma_i \in \mathscr{L}_{\text{PU}}$, two cases are possible, by Definition 5.10.2a.
   If $\forall v \in V_i, (\gamma_i(v).c = \bot \Rightarrow (\forall w \in \gamma_i(v).\mathcal{N}, \gamma_i(w).c \in \{0, \bot\}))$, then $\gamma_{i+1}(p).c = \gamma_{i+1}(q).c = 0$ (owing the fact that the c-clock values of all their non-$\bot$ neighbors are well computed according to their t-clock values, by Definition 5.10.1), and we are done.
   Otherwise, $\forall v \in V_i, (\gamma_i(v).c = \bot \Rightarrow (\exists w \in \gamma_i(v).\mathcal{N}, \gamma_i(w).c \neq \bot))$. So, $\exists x' \in \gamma_i(p).\mathcal{N}$ such that $\gamma_i(x').c \neq \bot \wedge \gamma_i(x').t = MinTime(p)$ in γ_i and $\exists y' \in \gamma_i(q).\mathcal{N}$ such that $\gamma_i(y').c \neq \bot \wedge \gamma_i(y').t = MinTime(q)$ in γ_i. By Definition 5.10.2c, $d_\beta(\gamma_i(x').t, \gamma_i(y').t) \leq \mu$ because they have neighbors whose c-variables equal $\bot$ (p and q, respectively). $\gamma_{i+1}(p).t = \gamma_i(x').t$ and $\gamma_{i+1}(q).t = \gamma_i(y').t$ since p and q execute $\mathcal{DSU}$-**J**-action. So $d_\beta(\gamma_{i+1}(p).t, \gamma_{i+1}(q).t) \leq \mu$.

■

Before going into details, we show the following theorem, which allows to simplify proofs and explanations.

Theorem 5.13

Let X be a closed set of configurations. Let ρ be any dynamic pattern.

$$\forall \gamma_i \in \mathcal{C}, (\exists \gamma_j \in X, \gamma_j \mapsto^{d,\rho} \gamma_i) \Leftrightarrow (\exists \gamma_k \in X, \gamma_k \mapsto^{d_{only},\rho} \gamma_i)$$

where $\mapsto^{d_{only},\rho}$ is the set of all ρ-dynamic steps containing no process activation.

Proof: Let $\gamma_i \in \mathcal{C}$ such that $\gamma_j \mapsto^{d,\rho} \gamma_i$ with $\gamma_j \in X$. If $\gamma_j \mapsto^{d_{only},\rho} \gamma_i$, we are done. Otherwise, let A be the non-empty subset of processes that are activated in $\gamma_j \mapsto^{d,\rho} \gamma_i$. There exists $\gamma_j \mapsto^{s} \gamma_u$, where A is activated. As X is closed, $\gamma_u \in X$. Moreover, $\forall x \in G_j \cap G_i$, $x \in G_u$ (since $G_u = G_j$) and $\gamma_u(x) = \gamma_i(x)$. Let $\gamma_u \mapsto^{d_{only},\rho} \gamma_k$ such that $G_k = G_i$. $\forall x \in G_j \cap G_i$, $x \in G_k$ (since $G_k = G_i$) and $\gamma_k(x) = \gamma_u(x) = \gamma_i(x)$. Moreover, $\forall x \in G_i \setminus G_j$, $x \in G_k$ (since $G_k = G_i$) and $\gamma_k(x) = \gamma_i(x)$ because in both cases, x is in bootstate. Hence, $\gamma_k = \gamma_i$, and we are done.

The second part of the assertion is trivial since, by definition, $\mapsto^{d_{only},\rho} \subseteq \mapsto^{d,\rho}$. ■

Lemma 5.22

Let $\gamma_i \in \mathscr{L}_{\text{SU}}$ and $\gamma_i \mapsto^{d,\mathsf{BULCC}}_{\mathcal{DSU}} \gamma_{i+1}$. Then, $\gamma_{i+1} \in \mathscr{L}_{\text{PU}}$.

Proof: By Theorem 5.13 and as $\mathscr{L}_{\text{SU}}$ is closed (Lemma 5.18), we can assume, without the loss of generality that, no process moves during $\gamma_i \mapsto^{d,\mathsf{BULCC}}_{\mathcal{DSU}} \gamma_{i+1}$. Then, the lemma is obvious by Definition of BULCC, Remark 5.5, and Definitions 5.9-5.10. ■

Lemma 5.23 (*Safety of SP_{PU} in $\mathcal{E}^0_{\mathcal{DSU}}(\mathscr{L}_{\text{PU}})$*)

Every execution $e \in \mathcal{E}^0_{\mathcal{DSU}}(\mathscr{L}_{\text{PU}})$ satisfies the safety of SP_{PU}.

Proof: Let $\gamma_i \in \mathscr{L}_{\text{PU}}$. If $\alpha \leq 3$, then by definition, for every two neighbors p and q in γ_i, we have $\big(\gamma_i(p).c \neq \bot \wedge \gamma_i(q).c \neq \bot\big) \Rightarrow d_\alpha(\gamma_i(p).c, \gamma_i(q).c) \leq 1$.

Assume now that $\alpha > 3$. By Definition 5.10.2b, for every two neighbors p and q in γ_i we have $\big(\gamma_i(p).c \neq \bot \wedge \gamma_i(q).c \neq \bot\big) \Rightarrow d_\beta(\gamma_i(p).t, \gamma_i(q).t) \leq \mu$. Furthermore, for every process p, $\gamma_i(p).c \neq \bot \Rightarrow \gamma_i(p).c = \left\lfloor \frac{\alpha}{\beta} \gamma_i(p).t \right\rfloor$. Hence, using Lemma 5.14 with $d = \mu < K$, $\forall p \in V_i$, $\forall q \in \gamma_i(p).\mathcal{N}$, $\big(\gamma_i(p).c \neq \bot \wedge \gamma_i(q).c \neq \bot\big) \Rightarrow d_\alpha(\gamma_i(p).c, \gamma_i(q).c) \leq 1$.

Finally, as the set $\mathscr{L}_{\text{PU}}$ is closed (Lemma 5.21), we are done. ■

By Lemmas 5.19 and 5.21, we have the following corollary.

Corollary 5.2

$\mathcal{DSU}$ converges from $\mathscr{L}_{\text{PU}}$ to $\mathscr{L}_{\text{WU}}$ in a finite time.

Lemma 5.24 (*Liveness of SP_{PU} in $\mathcal{E}^0_{\mathcal{DSU}}(\mathscr{L}_{PU})$*)

Every execution $e \in \mathcal{E}^0_{\mathcal{DSU}}(\mathscr{L}_{PU})$ satisfies the liveness of SP_{PU}.

Proof: Let $e = (\gamma_i)_{i\geq 0} \in \mathcal{E}^0_{\mathcal{DSU}}(\mathscr{L}_{PU})$. By Corollary 5.2, there exists $i \geq 0$ such that $\gamma_i \in \mathscr{L}_{WU}$. By Remarks 5.4, 5.9, and 5.10, we can apply Lemma 5.6: $p.t$ goes through each integer value between 0 and $\beta - 1$ infinitely often (in increasing order), for every process p. Hence, by Remark 5.6, for every process p, $p.c$ is incremented infinitely often and goes through each integer value between 0 and $\alpha - 1$ (in increasing order). ■

By Lemmas 5.21-5.24, we can deduce the following theorem.

Theorem 5.14

After one BULCC-dynamic step from a configuration of $\mathscr{L}_{SU}$, $\mathcal{DSU}$ immediately stabilizes to SP_{PU} by $\mathscr{L}_{PU}$.

Stabilization from $\mathscr{L}_{PU}$ to SP_{WU} by $\mathscr{L}_{WU}$ in at most one Round.

Lemma 5.25

$\mathcal{DSU}$ converges from $\mathscr{L}_{PU}$ to $\mathscr{L}_{WU}$ in finite time. The convergence time is at most one round.

Proof: The finite convergence time is proven by Corollary 5.2. Then, round complexity is trivial since every process in bootstate is continuously enabled to leave its bootstate by $\mathcal{DSU}$-**J**-action, and no process can set its c-clock to $\perp$ during a static step. ■

By Remarks 5.4, 5.9, and 5.10, results of Algorithm $\mathcal{WU}$ about $\mathscr{L}_{WU}$[5] also hold for Algorithm $\mathcal{DSU}$. Hence, follows.

Lemma 5.26 (*Closure and Correctness of $\mathscr{L}_{WU}$ under $\mathcal{DSU}$*)

$\mathscr{L}_{WU}$ is closed under $\mathcal{DSU}$, and for every execution $e \in \mathcal{E}^0_{\mathcal{DSU}}(\mathscr{L}_{WU})$ under $\mathcal{DSU}$, $SP_{WU}(e)$.

By Lemmas 5.25-5.26 and Theorem 5.14, follows.

Theorem 5.15

After one BULCC-dynamic step from a configuration of $\mathscr{L}_{SU}$, $\mathcal{DSU}$ stabilizes from $\mathscr{L}_{PU}$ to SP_{WU} by $\mathscr{L}_{WU}$ in finite time. The convergence time from $\mathscr{L}_{PU}$ to $\mathscr{L}_{WU}$ is at most one round.

[5]See Theorem 5.5 page 146.

Gradual Stabilization after one BULCC-Dynamic Step from $\mathscr{L}_{\text{SU}}$.

Lemma 5.27

The convergence time from $\mathscr{L}_{\text{WU}}$ to $\mathscr{L}_{\text{SU}}$ is at most $(\mu+1)\mathcal{D}_1+1$ rounds, where $\mathcal{D}_1$ is the diameter of the network after the dynamic step and μ is a parameter satisfying $\mu \geq \max(2, N)$.

Proof: Let $e = (\gamma_i)_{i\geq 0} \in \mathcal{E}^0_{\mathcal{DSU}}(\mathscr{L}_{\text{WU}})$. The behavior of the algorithm is similar to the one of $\mathcal{WU}$(Remarks 5.4, 5.9, and 5.10). By Theorem 5.7, within at most $\mu\mathcal{D}_1$ rounds the system reaches a configuration from which $\forall q \in p.\mathcal{N}$, $d_\beta(p.t, q.t) \leq 1$ forever, provided that no dynamic step occurs. By Lemma 5.11, each process increments its clock within at most $\mathcal{D}_1+1$ additional rounds, from that point the c-variables are well computed according to t-variables. Hence, in at most $(\mu+1)\mathcal{D}_1+1$ rounds, the system reaches $\mathscr{L}_{\text{SU}}$. ■

By Theorems 5.11-5.15 and Lemma 5.27, follows.

Theorem 5.16

$\mathcal{DSU}$ is gradually stabilizing under $(1, \mathsf{BULCC})$-dynamics for

$$(SP_{\text{PU}} \bullet 0, SP_{\text{WU}} \bullet 1, SP_{\text{SU}} \bullet (\mu+1)\mathcal{D}_1+2)$$

where $\mathcal{D}_1$ is the diameter of the network after the dynamic step and μ is a parameter satisfying $\mu \geq \max(2, N)$.

Theorem 5.17 establishes a bound on how many rounds are necessary to ensure that a given process increments its c-clock after the convergence to legitimate configurations w.r.t. SP_{SU}(resp. SP_{WU}).

Theorem 5.17

After convergence of $\mathcal{DSU}$ to $\mathscr{L}_{\text{WU}}$(resp. $\mathscr{L}_{\text{SU}}$), each process p increments its clock $p.c$at least once every $\mu\mathcal{D}_1+\frac{\beta}{\alpha}$ rounds (resp. $\mathcal{D}_1+\frac{\beta}{\alpha}$ rounds), where $\mathcal{D}_1$ is the diameter of the network after the dynamic step, and α, μ, and β are parameters respectively satisfying $\alpha \geq 2$, $\mu \geq \max(2, N)$, $\beta > \mu^2$, and β is multiple of α.

Proof: By Remarks 5.4, 5.9, and 5.10, we can use results on $\mathcal{WU}$ for $\mathcal{DSU}$. If $\mathcal{DSU}$ has converged to a configuration $\gamma \in \mathscr{L}_{\text{WU}}$, then $\gamma \in C_\mu$. So, by Lemma 5.10, after $\mu\mathcal{D}_1+\frac{\beta}{\alpha}$ rounds, p increments $p.t$ at least $\frac{\beta}{\alpha}$ times. Now, by Remark 5.6, if t-variable is incremented $\frac{\beta}{\alpha}$ times, c-variable is incremented once.

If $\mathcal{DSU}$ has converged to $\mathscr{L}_{\text{SU}}$, the result of Theorem 5.10 can be applied (Remarks 5.9 and 5.10). So, after $\mathcal{D}_1+\frac{\beta}{\alpha}$ rounds, p increments $p.c$ at least once. ■

5.7 Conclusion

Summary of Contributions. In this chapter, we have proposed a variant of self-stabilization, called gradual stabilization under (τ, ρ)-dynamics. An algorithm is gradually stabilizing under (τ, ρ)-dynamics if it is self-stabilizing and satisfied the following additional feature. From a legitimate configuration and after up to τ dynamic steps of type ρ, a gradually stabilizing algorithm first quickly recovers a configuration from which a minimum quality of service is satisfied. Then, it gradually converges to specifications offering stronger and stronger safety guarantees, until reaching a configuration from which its initial specification is satisfied, and where it is ready to achieve gradual convergence again if up to τ ρ-dynamic steps hit the system.

This new property is illustrated by considering three variants of unison problem, called strong, weak, and partial unison. Each process should maintain a local periodic clock of period $\alpha \geq 2$ and regularly increment it. The safety of strong unison requires that at most two consecutive value of clock exists at any step of the execution. Weak unison only requires that the difference between the clocks of two neighbors is at most one increment. Finally, we have defined the partial unison as a property dedicated to dynamic systems: only the difference of clocks between two neighbors that were present in the network before the dynamic step is constrained to be at most one increment.

We have proposed a gradually stabilizing algorithm under $(1, \mathsf{BULCC})$-dynamics, denoted $\mathcal{DSU}$, for arbitrary anonymous network (initially connected), designed in the locally shared memory model, and assuming the distributed unfair daemon. After a BULCC-dynamic step from a configuration satisfying the strong unison, $\mathcal{DSU}$ immediately satisfies the partial unison, then in one round the weak unison. It finally converges to strong unison in $(\mu+1)\mathcal{D}_1+2$ rounds, where μ is a parameter greater or equal than $\max(2, N)$, $\mathcal{D}_1$ is the diameter of the network after the dynamic step, and N is a bound on the number of processes in the network at any time of the execution.

A BULCC-dynamic step contains a finite number of topological changes such that, after such a step, the network:

1. contains at most N processes,
2. is connected,
3. if $\alpha > 3$, every process joining the system should be linked to at least one process that was already in the system before the dynamic step, except if all those processes have left the system.

Condition 1 is necessary to have finite periodic clocks in $\mathcal{DSU}$. We have shown that condition 2 is necessary. Finally, we have shown that condition 3 is necessary for our purposes when $\alpha > 5$. We have exhibited pathological cases for $\alpha = 4$ and $\alpha = 5$ if condition 3 is not satisfied.

Perspectives. The apparent seldomness of superstabilizing solutions for non-static problems, such as unison, may suggest the difficulty of obtaining such a strong property and if so, make our notion of gradual stabilization very attractive compared to

merely self-stabilizing solutions. For example, in our unison solution, gradual stabilization ensures that processes remain "almost" synchronized during the convergence phase started after one BULCC-dynamic step. Hence, it is worth investigating whether this new paradigm can be applied to other, in particular non-static, problems.

Concerning our unison algorithm, the graceful recovery after one dynamic step comes at the price of slowing down the clock increments. The question of limiting this drawback remains open.

Finally, it would be interesting to address in future work gradual stabilization for non-static problems in context of more complex dynamic patterns.

Chapter 6

Concurrency in Local Resource Allocation

"He who controls the spice controls the universe."

— Frank Herbert, *Dune*

Contents

6.1 Introduction

In this chapter, we consider *resource allocation* problems, *i.e.*, problems were some resources (*e.g.*, printers, files, memory) are shared among processes. Resource allocation problems consist in managing the access to ressources according to some usage rules. The portion of code that manages the access of a process to its allocated resources is called *critical section*.

Mutual exclusion [Dij65, Lam74] is a fundamental resource allocation problem, which consists in managing fair access of all (requesting) processes to a unique non-shareable reusable resource. This problem is inherently sequential, as no two processes should access this resource concurrently.

There are many other resource allocation problems which, in contrast, allow several resources to be accessed simultaneously. In those problems, parallelism on access to resources may be restricted by some of the following conditions:

1. The maximum number of resources that can be used concurrently, *e.g.*, the *ℓ-exclusion* problem [FLBB79], $\ell \geq 2$, is a generalization of the mutual exclusion problem which allows use of ℓ identical copies of a non-shareable reusable resource among all processes, instead of only one, as standard mutual exclusion. In other words, up to ℓ processes can concurrently execute their critical section.

2. The maximum number of resources a process can use simultaneously, *e.g.*, the *k-out-of-ℓ-exclusion* problem [Ray91], $1 \leq k \leq \ell$, is a generalization of ℓ-exclusion where a process can request for up to k resources simultaneously.

3. Some topological constraints, *e.g.*, in the *dining philosophers* problem [Dij78], two neighbors cannot use their common resource simultaneously.

For efficiency purposes, algorithms solving such problems must be as parallel as possible, *i.e.*, must allow as many processes in critical section concurrently as possible. As a consequence, these algorithms should be, in particular, evaluated at the light of the level of concurrency they permit, and this level of concurrency should be captured by a dedicated property. However, most of the resource allocation problems are specified in terms of safety and liveness properties only, *i.e.*, most of them include no property addressing concurrency performances, *e.g.*, [BPV04, CDP03, GH07, Hua00, NA02].

In this chapter, we especially focus on the concurrency level in resource allocation problems.

6.1.1 Related Work

As quoted by Fischer et al. [FLBB79], specifying resource allocation problems without including a property of concurrency may lead to degenerated solutions, *e.g.*, any mutual exclusion algorithm realizes safety and fairness of ℓ-exclusion. However, at most one process is in critical section at any time.

To address this issue, Fischer et al. [FLBB79] proposed an *ad hoc* property to capture concurrency in ℓ-exclusion problem. This property is called *avoiding ℓ-deadlock* and is informally defined as follows: "if fewer than ℓ processes are executing their critical section, then it is possible for another process to enter its critical section, even though no process leaves its critical section in the meantime."

Some other properties, inspired from the avoiding ℓ-deadlock property, have been proposed to capture the level of concurrency in other resource allocation problems, *e.g.*, k-out-of-ℓ-exclusion [DHV03] and committee coordination [BDP11]. However, until now, all existing properties of concurrency are specific to a particular problem, *e.g.*, the avoiding ℓ-deadlock property cannot be applied to committee coordination.

In this chapter, we propose to generalize the avoiding ℓ-deadlock property to any resource allocation problem.

Then, we focus our study on the *Local Resource Allocation (LRA)* problem, defined by Cantarell et al. [CDP03]. LRA is a generalization of resource allocation problems in which resources are shared among neighboring processes. Dining philosophers, local readers-writers, local mutual exclusion, and local group mutual exclusion are particular instances of LRA. In contrast, local ℓ-exclusion and local k-out-of-ℓ-exclusion cannot be expressed with LRA although they also deal with neighboring resource sharing.

We aim to design a stabilizing solution to LRA achieving a high level of concurrency. There exist many algorithms for particular instances of the LRA problem. Many of these solutions have been proven to be self-stabilizing, *e.g.*, [BPV04, CDP03, GH07, Hua00, NA02].

In [BPV04], Boulinier et al. propose a self-stabilizing unison (*i.e.*, clock synchronization) algorithm which allows to solve local mutual exclusion, local group mutual exclusion, and local readers-writers problem. In [NA02], Nesterenko and Arora propose self-stabilizing algorithms for the solving the local mutual exclusion, dining philosophers, and drinking philosophers problems. There are also many self-stabilizing algorithms for local mutual exclusion, *e.g.*, [GH07, Hua00].

In [CDP03], Cantarell et al. generalize the above problems by introducing the LRA problem. They also propose a self-stabilizing algorithm for that problem. To the best of our knowledge, no other paper deals with the general instance of LRA. Moreover, none of the aforementioned papers (especially [CDP03]) consider the maximal concurrency issue.

Finally, note that there exist weaker versions of the LRA problem, such as the (local) *conflict managers* proposed in [GT07] where the fairness is replaced by a progress property, *i.e.*, it does not require that any requesting process eventually execute its critical section but only that at least one of the requesting processes is satisfied.

6.1.2 Contributions

In this chapter, we first propose a generalization of avoiding ℓ-deadlock to any resource allocation problems. We call this new property the *maximal concurrency* (Section 6.3).

We show that maximal concurrency cannot be achieved in a wide class of instances of

the LRA problem (Section 6.4). This impossibility result is mainly due to the fact that fairness of LRA and maximal concurrency are incompatible properties: it is impossible to implement an algorithm achieving both properties together.

As unfair resource allocation algorithms are clearly unpractical, we propose to weaken the property of maximal concurrency (Section 6.5). We call *partial concurrency* this weaker version of maximal concurrency. The goal of *partial concurrency* is to capture the maximal level of concurrency that can be obtained in any instance of the LRA problem without compromising fairness.

Then, we propose in Section 6.6 a LRA algorithm achieving a strong form of partial concurrency in bidirectional identified networks of arbitrary topology. As additional feature, this algorithm is *snap-stabilizing* [BDPV07], *i.e.*, after transient faults cease, a snap-stabilizing algorithm *immediately* resumes correct behavior, without external intervention. More precisely, a snap-stabilizing algorithm guarantees that any computation (here, any request to access some shared resources) started after the faults cease will operate correctly. In our knowledge, it is the first snap-stabilizing algorithm solving LRA or any particular instance of LRA.

An implementation of this algorithm requires $\Theta\big(\log n + \log(\max\{|\mathcal{R}_p| : p \in V\})\big)$ bits per processes where $\mathcal{R}_p$ is the set of resources that can be requested and used by a process p. Futhermore, the request of a process is satisfied in $O(nC)$ rounds, where C is an upper bound on the execution time of a critical section and n is the number of processes.

These results appear in the proceedings of the 3rd International Conference on Networked Systems (NETYS 2015) [ADD15], in the Journal of Parallel and Distributed Computing [ADD17b], and in the proceedings of the 18èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications (ALGOTEL 2016) [ADD16b].

6.2 Preliminaries

We first detail the context (Section 6.2.1). Then we formally define the local resource allocation algorithm (Section 6.2.2).

6.2.1 Context

We consider static bidirectionnal connected and identified networks of arbitrary topology. We assume the locally shared memory model under a distributed weakly fair daemon (see Section 2.6).

We denote by $\mathcal{R}_p$ the set of resources that can be requested (and used) by process p. We consider algorithms interacting with their environment. More precisely, the user or the application on the process requires the access to some shared resources through some inputs of the algorithm.

6.2.2 Snap-stabilizing Local Resource Allocation

In the *Local Resource Allocation (LRA)* problem [CDP03] each process requests at most one resource at a time. The problem is based on the notion of compatibility between

resources: two resources X and Y are said to be *compatible*, and we denote $X \rightleftharpoons Y$, if two neighbors can concurrently access them. Otherwise, X and Y are said to be *conflicting*, and we denote $X \not\rightleftharpoons Y$. Notice that $\rightleftharpoons$ is a symmetric relation.

The local resource allocation problem consists in ensuring that every process which requires a resource r eventually accesses r while no other conflicting resource is currently used by a neighbor. In contrast, there is no restriction for concurrently allocating the same resource to any number of processes that are not neighbors.

Notice that the case where there are no conflicting resources is trivial: a process can always use a resource whatever the state of its neighbors. So, from now on, we will always assume that there exists at least one conflict, *i.e.*, there are (at least) two neighbors p, q and two resources X, Y such that $X \in \mathcal{R}_p, Y \in \mathcal{R}_q$ and $X \not\rightleftharpoons Y$. This also means that any network considered from now on contains at least two processes.

Specifying the relation $\rightleftharpoons$, it is possible to define some classic resource allocation problems in which the resources are shared among neighboring processes.

Example 1: Local Mutual Exclusion. In the *local mutual exclusion* problem, no two neighbors can concurrently access the unique resource. So there is only one resource X common to all processes and $X \not\rightleftharpoons X$.

Example 2: Local Readers-Writers. In the *local readers-writers* problem, the processes can access a file in two different modes: a read access (the process is said to be a *reader*) or a write access (the process is said to be a *writer*). A writer must access the file in local mutual exclusion, while several reading neighbors can concurrently access the file. We represent these two access modes by two resources at every process: R for a "read access" and W for a "write access." Then, $\text{R} \rightleftharpoons \text{R}$, but $\text{W} \not\rightleftharpoons \text{R}$ and $\text{W} \not\rightleftharpoons \text{W}$.

Example 3: Local Group Mutual Exclusion. In the *local group mutual exclusion* problem, there are several resources $r_0, r_1, \ldots, r_k$ shared between the processes. Two neighbors can access concurrently the same resource but cannot access different resources at the same time. Then:

$$\forall i \in \{0,\ldots,k\}, \forall j \in \{0,\ldots,k\}, \begin{cases} r_i \rightleftharpoons r_j & \text{if } i = j, \\ r_i \not\rightleftharpoons r_j & \text{otherwise.} \end{cases}$$

Snap-stabilizing LRA Specification. Let Alg be a distributed algorithm. As stated in Section 2.7, snap-stabilization has initially been defined as follows: Alg is *snap-stabilizing* w.r.t. some specification SP if starting from any arbitrary configuration, all its executions satisfy SP.

Of course, not all specifications — in particular their safety part — can be satisfied when considering a system which can start from an arbitrary configuration. Actually, snap-stabilization's notion of safety is *user-centric*: when the user initiates a computation, then the computed result should be correct. So, we express a problem using a *guaranteed service specification* [AD14]. Such a specification consists in specifying three properties

related to the computation start, computation end, and correctness of the delivered result. (In the context of LRA, this latter property will be referred to as "resource conflict freedom.")

To formally define the guaranteed service specification of the local resource allocation problem, we need to introduce the following four predicates, where p is a process, r is a resource, and $e = (\gamma_i)_{i\geq 0}$ is an execution:

- $Req(\gamma_i, p, r)$ means that an application at p requests for r in configuration γ_i. We assume that the application cannot change its request, *i.e.*, if $Req(\gamma_i, p, r)$ holds then $Req(\gamma_j, p, r)$ holds for any $j \geq i$ (at least) until p accesses r.
- $Start(\gamma,\gamma_{i+1}, p, r)$ means that p starts a computation to access r in $\gamma_i \mapsto \gamma_{i+1}$.
- $Result(\gamma_i \dots \gamma_j, p, r)$ means that p obtains access to r in $\gamma_{i-1} \mapsto \gamma_i$ and p ends the computation in $\gamma_j \mapsto \gamma_{j+1}$. Notably, p released r between γ_i and γ_j.
- $NoConflict(\gamma_i, p)$ means that, if a resource is allocated to p in γ_i, then none of its neighbors is using a conflicting resource.

These predicates will be instantiated with the variables of the local resource allocation algorithm, see Section 6.6.1 (p. 188). Below, we define the guaranteed service specification of LRA, denoted SP_{LRA}.

Definition 6.1 (*Guaranteed Service Local Resource Allocation*)

Let Alg be an algorithm. An execution $e = (\gamma_i)_{i\geq 0}$ of Alg satisfies the guaranteed service specification of LRA, noted SP_{LRA}, if the three following properties hold:

- *Resource Conflict Freedom:* If a process p starts a computation to access a resource, then there is no conflict involving p during the computation, *i.e.*, $\forall k \geq 0$, $\forall k' > k$, $\forall p \in V$, $\forall r \in \mathcal{R}_p$,

$$\begin{array}{c}\big[Result(\gamma_k \dots \gamma_{k'}, p, r) \wedge \big(\exists l < k, Start(\gamma_l, \gamma_{l+1}, p, r)\big)\big] \\ \Rightarrow \ \big[\forall i \in \{k, \dots, k'\}, NoConflict(\gamma_i, p)\big]\end{array}$$

- *Computation Start:* If an application at process p requests resource r, then p eventually starts a computation to obtain r, *i.e.*, $\forall k \geq 0$, $\forall p \in V$, $\forall r \in \mathcal{R}_p$,

$$\big[\exists l > k, Req(\gamma_l, p, r) \Rightarrow Start(\gamma_l, \gamma_{l+1}, p, r)\big]$$

- *Computation End:* If process p starts a computation to obtain resource r, the computation eventually ends (in particular, p obtained r during the computation), *i.e.*, $\forall k \geq 0$, $\forall p \in V$, $\forall r \in \mathcal{R}_p$,

$$Start(\gamma_k, \gamma_{k+1}, p, r) \Rightarrow \big[\exists l > k, \exists l' > l, Result(\gamma_l \dots \gamma_{l'}, p, r)\big]$$

Thus, an algorithm Alg is snap-stabilizing w.r.t. SP_{LRA} (*i.e.*, snap-stabilizing for LRA) if starting from any arbitrary configuration, all its executions satisfy SP_{LRA}.

6.3 Maximal Concurrency

In [FLBB79], authors propose a concurrency property *ad hoc* to the ℓ-exclusion problem. We now define the *maximal concurrency*, which generalizes the definition of [FLBB79] to any resource allocation problem.

6.3.1 Definition

Informally, maximal concurrency can be defined as follows: if there are processes that can access resources they are requesting without violating the safety of the considered resource allocation problem, then at least one of them should eventually satisfies its request, even if no process releases the resource(s) it holds meanwhile.

For any configuration γ, we define three sets of processes:

- $P_{CS}(\gamma)$ is the set of processes that are executing their critical section in γ, *i.e.*, the set of processes holding resources in γ.

- $P_{Req}(\gamma)$ is the set of requesting processes that are not in critical section in γ, *i.e.*, their request is not yet satisfied in γ.

- $P_{Free}(\gamma) \subseteq P_{Req}(\gamma)$ is the set of requesting processes that can access their requested resource(s) in γ without violating the safety of the considered resource allocation problem.

For any execution $(\gamma_i)_{i\geq 0}$, let

$$\begin{array}{lcl} ContinuousCS(\gamma_i \dots \gamma_j) & \equiv & \forall k \in \{i+1, \dots, j\},\ P_{CS}(\gamma_{k-1}) \subseteq P_{CS}(\gamma_k) \\ NoRequest(\gamma_i \dots \gamma_j) & \equiv & \forall k \in \{i+1, \dots, j\},\ P_{Req}(\gamma_k) \subseteq P_{Req}(\gamma_{k-1}) \end{array}$$

$ContinuousCS(\gamma_i \dots \gamma_j)$ (respectively, $NoRequest(\gamma_i \dots \gamma_j)$) means that no resource is released (respectively, no new request occurs) between γ_i and γ_j. Notice that for any $i \geq 0$, $ContinuousCS(\gamma_i)$ and $NoRequest(\gamma_i)$ trivially hold.

Let $e = (\gamma_i)_{i\geq 0}$, $k \geq 0$ and $t \geq 0$. The function $R(e, k, t)$ is defined if and only if the execution $(\gamma_i)_{i\geq k}$ contains at least t rounds. If it is defined, the function returns $x \geq k$ such that the execution factor $\gamma_k \dots \gamma_x$ contains exactly t rounds, *i.e.*, the t^{th} round ends in γ_x.

Definition 6.2 (*Maximal Concurrency*)

A resource allocation algorithm ALG is *maximal concurrent* in a network $\mathcal{G} = (V, E)$ if and only if

- *No Deadlock:* For every configuration γ such that $P_{Free}(\gamma) \neq \emptyset$, there exists a configuration γ' and a step $\gamma \mapsto \gamma'$ such that

$$ContinuousCS(\gamma\gamma') \wedge NoRequest(\gamma\gamma')$$

- *No Livelock:* There exists a number of rounds N such that for every execution $e = (\gamma_i)_{i\geq 0}$ and for every index $i \geq 0$, if $R(e, i, N)$ exists, then

$$\begin{aligned}\big(&NoRequest(\gamma_i \dots \gamma_{R(e,i,N)}) \wedge ContinuousCS(\gamma_i \dots \gamma_{R(e,i,N)}) \wedge P_{Free}(\gamma_i) \neq \emptyset\big)\\ &\Rightarrow (\exists k \in \{i, \dots, R(e, i, N) - 1\}, \exists p \in V, p \in P_{Free}(\gamma_k) \cap P_{CS}(\gamma_{k+1}))\end{aligned}$$

No Deadlock ensures that whenever a request can be satisfied, the algorithm is not deadlocked and can still execute some step, even if no resource is released and no new request happens.

No Livelock assumes that there exists a number of round N (which depends on the complexity of the algorithm, and henceforth on the network dimensions) such that: if during an execution, there exists some requests that can be satisfied, then at least one of them should be satisfied within N rounds, even if no resource is released and no new request happens meanwhile. Notice that the mention "no new request happens meanwhile" ensures that N uniquely depends on the algorithm and the network; if not, N would also depend on the scheduling of the requests.

6.3.2 Alternative Definition

We now provide an alternative definition of maximal concurrency: instead of constraining P_{Free} to decrease every N rounds during which there is neither new request, nor critical section exit, it expresses that P_{Free} becomes empty after enough rounds in such a situation.

We introduce first some notations: let $e = (\gamma_i)_{i\geq 0}$ be an execution and $i \geq 0$ be the index of configuration γ_i. We note $endCS(e, i)$ (respectively, $reqUp(e, i)$) the last configuration index such that no resource is released (respectively, no new request occurs and no resource is released) during the execution factor $\gamma_i \dots \gamma_{endCS(e,i)}$ (respectively, $\gamma_i \dots \gamma_{reqUp(e,i)}$). Formally,

$$\begin{aligned} endCS(e, i) &= \max\{j \geq i : ContinuousCS(\gamma_i \dots \gamma_j)\}\\ reqUp(e, i) &= \max\{j \geq i : NoRequest(\gamma_i \dots \gamma_j) \wedge j \leq endCS(e, i)\}\end{aligned}$$

Note that $endCS(e, i)$ is always defined (for any e and any i) since $ContinuousCS(\gamma_i)$ holds and any critical section is assume to be finite. Consequently, $reqUp(e, i)$ is always

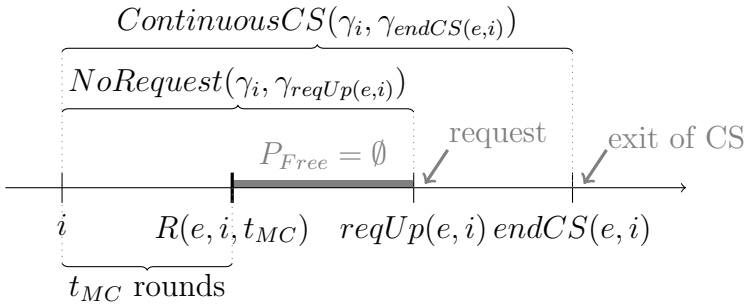


Figure 6.1 – Illustration of Definition 6.3

defined, since the set $\{j \geq i : NoRequest(\gamma_i \dots \gamma_j) \wedge j \leq endCS(e, i)\}$ is not empty and bounded by $endCS(e, i)$.

Definition 6.3 (*Maximal Concurrency*)

A resource allocation algorithm ALG is *maximal concurrent* in a network $\mathcal{G} = (V, E)$ if and only if

- *No Deadlock:* For every configuration γ such that $P_{Free}(\gamma) \neq \emptyset$, there exists a configuration γ' and a step $\gamma \mapsto \gamma'$ such that

$$ContinuousCS(\gamma\gamma') \wedge NoRequest(\gamma\gamma')$$

- *No Livelock:* There exists a number of rounds t_{MC} such that for every execution $e = (\gamma_i)_{i \geq 0}$ and for every index $i \geq 0$, if $R(e, i, t_{MC})$ exists, then

$$R(e, i, t_{MC}) \leq reqUp(e, i) \Rightarrow P_{Free}(\gamma_{R(e,i,t_{MC})}) = \emptyset$$

No Deadlock is identical in Definitions 6.2 and 6.3. However, *No Livelock* assumes now that there exists a (greater) number of rounds t_{MC} such that if no resource is released and no new request happens during t_{MC} rounds, then the set P_{Free} becomes empty. As in the former definition, t_{MC} depends on the complexity of the algorithm. Definition 6.3 is illustrated by Figure 6.1.

Lemma 6.1

Definition 6.2 and Definition 6.3 are equivalent.

Proof: Note first that the *No Deadlock* part is identical in both definitions.

Consider now the *No Livelock* part: If Definition 6.2 holds, then Definition 6.3 holds by letting $t_{MC} = n \times N$; if Definition 6.3 holds, then Definition 6.2 holds by letting $N = t_{MC}$. ■

Using Definition 6.3, remark that an algorithm is not maximal concurrent in a network $\mathcal{G} = (V, E)$ if and only if:

- either the property *No Deadlock* is violated, *i.e.*, there exists a configuration γ such that $P_{Free}(\gamma) \neq \emptyset$ and for any configuration γ' such that $ContinuousCS(\gamma\gamma') \wedge NoRequest(\gamma\gamma')$, there is no possible step of the algorithm from γ to γ' ($\gamma \not\mapsto \gamma'$);
- or the property *No Livelock* is violated: for every $t > 0$, there exists an execution $e = (\gamma_i)_{i \geq 0}$ and an index $i \geq 0$ such that $R(e, i, t)$ is defined, $R(e, i, t) \leq reqUp(e, i)$, and $P_{Free}(\gamma_{R(e,i,t)}) \neq \emptyset$.

6.3.3 Instantiations

The examples below show the versatility of our property: we instantiate the set P_{Free} according to the considered problem. Note that the first problem is local, whereas others are not.

Below, we denote by $\gamma(p).req$ the resource requested/used by process p in configuration γ. If p neither requests nor uses any resource, then $\gamma(p).req = \perp$, where $\perp$ is compatible with every resource.

Example 1: Local Resource Allocation. In the local resource allocation problem, a requesting process is allowed to enter its critical section if all its neighbors in critical section are using resources which are compatible with its requested resource. Hence,

$$P_{Free}(\gamma) = \{p \in P_{Req}(\gamma) : \forall q \in p.\mathcal{N}, (q \in P_{CS}(\gamma) \Rightarrow \gamma(q).req \rightleftharpoons \gamma(p).req)\}$$

Example 2: ℓ-Exclusion. The ℓ-exclusion problem [FLBB79] is a generalization of mutual exclusion, where up to $\ell \geq 1$ critical sections can be executed concurrently. Solving this problem allows the management of a pool of ℓ identical units of a non-sharable reusable resource. Hence,

$$P_{Free}(\gamma) = \begin{cases} \emptyset & \text{if } |P_{CS}(\gamma)| = \ell, \\ P_{Req}(\gamma) & \text{otherwise.} \end{cases}$$

Using this latter instantiation, we obtain a definition of maximal concurrency which is equivalent to the "avoiding ℓ-deadlock" property of Fischer et al. [FLBB79].

Example 3: k-out-of-ℓ Exclusion. The k-out-of-ℓ exclusion problem [DHV03] is a generalization of the ℓ-exclusion problem where each process can hold up to $k \leq \ell$ identical units of a non-sharable reusable resource. In this context, rather than being the resource(s) requested by process p, $\gamma(p).req$ is assumed to be the number of requested units, *i.e.*, $\gamma(p).req \in \{0, \ldots, k\}$. Let $Available(\gamma) = \ell - \sum_{p \in P_{CS}(\gamma)} \gamma(p).req$ be the number of available units. Hence,

$$P_{Free}(\gamma) = \{p \in P_{Req}(\gamma) : \gamma(p).req \leq Available(\gamma)\}$$

Using this latter instantiation, we obtain a definition of maximal concurrency which is equivalent to the "strict (k, ℓ)-liveness" property of Datta et al. [DHV03], which basically

means that if *at least one* request can be satisfied using the available resources, then eventually one of them is satisfied, even if no process releases resources in the meantime.

In [DHV03], the authors show the impossibility of designing a k-out-of-ℓ exclusion algorithm satisfying the strict (k, ℓ)-liveness. To circumvent this impossibility, they then propose a weaker property called "(k, ℓ)-liveness", which means that if *any* request can be satisfied using the available resources, then eventually one of them is satisfied, even if no process releases resources in the meantime. Despite this property is weaker than maximal concurrency, it can be expressed using our formalism as follows:

$$P_{Free}(\gamma) = \begin{cases} \emptyset & \text{if } \exists p \in P_{Req}(\gamma), \gamma(p).req > Available(\gamma), \\ P_{Req}(\gamma) & \text{otherwise.} \end{cases}$$

This might seem surprising, but observe that in the above formula, the set P_{Free} is distorted from its original meaning.

6.3.4 Strict (k, ℓ)-liveness versus Maximal Concurrency

As an illustrative example, we now show that the original definition of *strict* (k, ℓ)-*liveness* [DHV03] is equivalent to the instantiation of maximal concurrency we propose in Example 3 of the previous subsection.

In [DHV03], to introduce strict (k, ℓ)-liveness, the authors assume that a process can stay in critical section forever. Notice that this assumption is only used to define strict (k, ℓ)-liveness, critical sections are otherwise always assumed to be finite. Using this artifact, they express that a k-out-of-ℓ exclusion algorithm satisfies the *strict* (k, ℓ)-*liveness* in a network $\mathcal{G} = (V, E)$ as follows:

Let $P \subseteq V$ be the set of processes executing the critical section forever. Let $nbFree = \ell - \sum_{p \in P} p.req$. If there exists $p \in V$ such that p is requesting for $p.req \leq nbFree$ resources, then, eventually at least one requesting process (maybe p) enters the critical section.

Maximal Concurrency $\Rightarrow$ Strict (k, ℓ)-Liveness. Let Alg be a k-out-of-ℓ exclusion algorithm which is maximal concurrent in a network $\mathcal{G} = (V, E)$. Assume an execution starting in configuration γ such that there is a set P of processes executing the critical section forever from γ. Assume by contradiction that from γ, no requesting process ever enters the critical section although there exists a requesting process p such that $p.req \leq nbFree$.

As the number of processes is finite, the system eventually reaches a configuration γ' from which no new request ever occur. By *No Deadlock*, the execution from γ' is infinite. Moreover, the daemon being weakly fair, every round from γ' is finite. Now, by *No Livelock*, after a finite number of rounds, one process enters the critical section (*n.b.*, P_{Free} is not empty because of p), a contradiction. Hence, Alg satisfies the strict (k, ℓ)-liveness in $\mathcal{G}$.

¬ Maximal Concurrency ⇒ ¬ Strict (k, ℓ)-Liveness. Let ALG′ be a k-out-of-ℓ exclusion algorithm which is not maximal concurrent in a network $\mathcal{G} = (V, E)$. Assume first ALG′ does not satisfy *No Deadlock*: there exists a configuration γ such that $P_{Free}(\gamma) \neq \emptyset$ and for every configuration γ' such that $ContinuousCS(\gamma\gamma') \wedge NoRequest(\gamma\gamma')$, there is no possible step of the algorithm from γ to γ'. Assume an execution from γ where all critical sections are infinite and there is no new request. Then, the system is deadlocked and, consequently, no process of P_{Free} can enter the critical section. Now, P_{Free} is not empty. So, there exists a requesting process p such that $p.req \leq nbFree$. Moreover, only processes of P_{Free} can enter critical section without violating safety. Consequently, no process ever enter the critical section during this execution: ALG′ does not satisfy the strict (k, ℓ)-liveness in $\mathcal{G}$.

Finally, assume ALG′ violates *No Livelock*: for every $t > 0$, there exists an execution $e = (\gamma_i)_{i \geq 0}$ and an index $i \geq 0$ such that $R(e, i, t)$ is defined, $R(e, i, t) \leq reqUp(e, i)$, and $P_{Free}(\gamma_{R(e,i,t)}) \neq \emptyset$. So, it is possible to build an infinite execution e, where all critical sections are infinite, no new request happens, and P_{Free} is never empty. As there is no new request, P_{Free} is never empty, and the number of processes is finite, there is an infinite suffix s of e where no process leaves P_{Free} (*i.e.*, no process of P_{Free} enters critical section) although P_{Free} is not empty. In s, there exists a requesting process p such that $p.req \leq nbFree$ because P_{Free} is not empty, but no process ever enter the critical section because only processes of P_{Free} can enter critical section without violating safety. Hence, ALG′ does not satisfy the strict (k, ℓ)-liveness in $\mathcal{G}$.

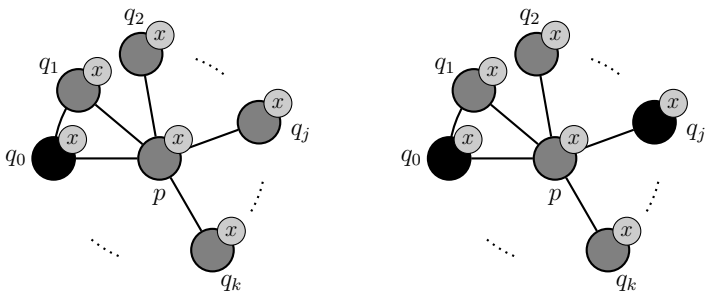
Hence, the original definition of strict (k, ℓ)-liveness [DHV03] is equivalent to the instantiation of maximal concurrency proposed in Example 3 of the previous subsection.

6.4 Maximal Concurrency versus Fairness

Maximal concurrency is achievable in ℓ-exclusion, see [FLBB79]. However, there exist problems where it is not possible to ensure the maximal degree of concurrency, *e.g.*, Datta et al. showed in [DHV03] that it is impossible to design a k-out-of-ℓ exclusion algorithm that satisfies the strict (k, ℓ)-liveness, which is equivalent to the maximal concurrency. Precisely, the impossibility proof shows that in this problem, fairness and maximal concurrency are incompatible properties. We now study the maximum degree of concurrency that can be achieved by a LRA algorithm.

6.4.1 Necessary Condition on Concurrency in LRA

Definition 6.4 below gives a definition of fairness classically used in resource allocation problems. Notably, *Computation Start* and *Computation End* properties of the LRA specification (see Definition 6.1) trivially implies this fairness property. Next, Lemma 6.2 is a technical result which will be used to show that there are (important) instances of the LRA problem for which it is impossible to design a maximal concurrent algorithm working in arbitrary networks (Theorem 6.1).



(a) γ_0 **(b)** γ_{i+1}

Figure 6.2 – Outline of the execution $(\gamma_i)_{i\geq 0}$ of the proof of Lemma 6.2 on the neighborhood of p. Black nodes are in critical section, gray nodes are requesting.

Definition 6.4 (*Fairness*)

Every time a process is (continuously) requesting some resource r, it eventually accesses r.

We recall that $\gamma(p).req$ denotes the resource requested/used by process p in configuration γ. If p neither requests nor uses any resource, then $\gamma(p).req = \perp$, where $\perp$ is compatible with every resource. We define the *conflicting neighborhood* of p in γ, denoted by $\gamma(p).\mathcal{CN}$, as follows: $\gamma(p).\mathcal{CN} = \{q \in p.\mathcal{N} : \gamma(p).req \not\simeq \gamma(q).req\}$. Note that if p is not requesting, then $\gamma(p).\mathcal{CN} = \emptyset$.

Below we consider any instance $\mathscr{I}$ of the LRA problem, where every process can request the same set of resources $\mathcal{R}$ (*i.e.*, $\forall p \in V, \mathcal{R}_p = \mathcal{R}$) and $\exists x \in \mathcal{R}$ such that $x \not\simeq x$. Notice that the local mutual exclusion and the local readers-writers problem belong to this class of LRA problems.

Lemma 6.2

For any algorithm solving $\mathscr{I}$ in a network $\mathcal{G} = (V, E)$, if $|V| > 1$, then for any process p, there exists an execution $e = (\gamma_i)_{i\geq 0}$, with configuration γ_t, $t \geq 0$, and a process $q \in \gamma_t(p).\mathcal{CN}$ such that

1. $p.\mathcal{N} \backslash (\{q\} \cup q.\mathcal{N}) = \gamma_t(p).\mathcal{CN} \backslash \big(\{q\} \cup \gamma_t(q).\mathcal{CN}\big) = P_{Free}(\gamma_t)$
2. *and for every execution $e' = (\gamma'_i)_{i\geq 0}$ which shares the same prefix as e between γ_0 and γ_t (i.e., $\forall i \in \{0, ..., t\}, \gamma_i = \gamma'_i$),*

$$\forall t' \in \{t, ..., reqUp(e', t)\}, P_{Free}(\gamma_t) = P_{Free}(\gamma'_{t'})$$

Proof: Consider any algorithm solving $\mathscr{I}$ in a network $\mathcal{G} = (V, E)$ with $|V| > 1$. Let $p \in V$.

First, consider the case when p has a unique neighbor q. Assertion 1 trivially holds for any configuration γ_t since $p.\mathcal{N} \backslash (\{q\} \cup q.\mathcal{N}) = \gamma_t(p).\mathcal{CN} \backslash \big(\{q\} \cup \gamma_t(q).\mathcal{CN}\big) = \emptyset$: Let $t = 0$ and γ_0 be a configuration such that p is requesting a resource, q holds a resource conflicting with the resource requested by p, and no other process is either requesting

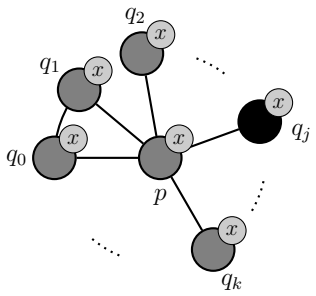


Figure 6.3 – Neighborhood of p in configuration γ'_{i+2} in the proof of Lemma 6.2.

or executing its critical section. In this case, $P_{Free}(\gamma_0) = \emptyset$ and $P_{Req}(\gamma_0) = \{p\}$. Then, for every possible execution from γ_0, as long as q holds its resource and no new request occurs, P_{Free} remains empty, which proves Assertion 2.

Then, we assume that p has at least two neighbors. We note $p.\mathcal{N}$ as $\{q_0, ..., q_k\}$ with $k \geq 1$. We fix γ_0 such that

- q_0 holds some resource x such that x is conflicting with x,
- p requests resource x,
- for all $j \in \{1, ..., k\}$, q_j requests resource x,
- no other process is either requesting or executing critical section,

namely:

$$\begin{array}{lcl} P_{Free}(\gamma_0) & = & \gamma_0(p).\mathcal{CN} \backslash \big(\{q_0\} \cup \gamma_0(q).\mathcal{CN} \big) = p.\mathcal{N} \backslash (\{q_0\} \cup q_0.\mathcal{N}) \\ P_{Req}(\gamma_0) & = & \{p\} \cup \{q_j : j \in \{1, ..., k\}\} \end{array}$$

See γ_0 in Figure 6.2a.

Again, if $P_{Free}(\gamma_0) = \emptyset$, then we let $t = 0$ and Assertion 1 holds. Moreover, in this case, every q_j, with $j \in \{1, ..., k\}$ is a neighbor of q_0. Hence, for any possible execution from γ_0, as long as q_0 holds x and no new request occurs, P_{Free} remains empty: Assertion 2 holds.

Assume now that $P_{Free}(\gamma_0) \neq \emptyset$. We build an execution by letting the algorithm executes, while maintaining q_0 in critical section and trigerring no new request (this is possible by the *No Deadlock* property). If no neighbor of p ever exits from P_{Free}, Assertions 1 and 2 are both satisfied.

Otherwise, let $i > 0$ and $j \in \{1, ..., k\}$ such that q_j is the first neighbor of p to exit P_{Free} and $\gamma_i \mapsto \gamma_{i+1}$ is the first step where q_j exits from P_{Free}. (See Configuration γ_{i+1} on Figure 6.2b.) We replace step $\gamma_i \mapsto \gamma_{i+1}$ by two steps $\gamma_i \mapsto \gamma'_{i+1} \mapsto \gamma'_{i+2}$ such that:

- q_j leaves $P_{Free}(\gamma_i)$ and has access to x (by assumption) in $\gamma_i \mapsto \gamma'_{i+1}$,
- q_0 releases its critical section in $\gamma_i \mapsto \gamma'_{i+1}$ and requests again x in $\gamma'_{i+1} \mapsto \gamma'_{i+2}$.

Configuration γ'_{i+2} is shown in Figure 6.3. Hence,

$$\begin{array}{lcl} P_{Req}(\gamma'_{i+2}) & = & \{p\} \cup \{q_l : l \neq j \wedge l \in \{0, ..., k\}\} \\ P_{Free}(\gamma'_{i+2}) & = & \gamma'_{i+2}(p).\mathcal{CN} \backslash \big(\{q_j\} \cup \gamma'_{i+2}(q_j).\mathcal{CN} \big) = p.\mathcal{N} \backslash (\{q_j\} \cup q_j.\mathcal{N}) \end{array}$$

So, in γ'_{i+2} the system is in a situation similar to γ_0. If this scenario is repeated indefinitely, the algorithm never satisfies the request of p, contradicting the fairness of

the LRA specification. Hence, there exists a configuration γ_t, $t \geq 0$ after which P_{Free} remains equal to $\gamma_t(p).\mathcal{CN} \backslash \big(\{q_l\} \cup \gamma_t(q_l).\mathcal{CN} \big) = p.\mathcal{N} \backslash (\{q_l\} \cup q_l.\mathcal{N})$ (this proves Assertion 1) and constant for some $q_l \in p.\mathcal{N}$, until q_l releases its resource or some new request occurs (this proves Assertion 2). ■

6.4.2 Impossibility Result

Using Lemma 6.2, we show that it is impossible to solve the instances of LRA problem defined in Section 6.4.1 with a maximal concurrent algorithm in arbitrary networks.

Theorem 6.1

It is impossible to design a maximal concurrent algorithm solving $\mathscr{I}$ in every network.

Proof: Assume by contradiction that there is a maximal concurrent algorithm solving $\mathscr{I}$ in every network. Consider a network (of at least two processes) which contains a process p, such that $\forall q \in p.\mathcal{N}$, $p.\mathcal{N} \backslash (\{q\} \cup q.\mathcal{N}) \neq \emptyset$. (Take for instance a star network where p is at the center.)

From Lemma 6.2, there exists $e = (\gamma_i)_{i \geq 0}$ with a configuration γ_t, $t \geq 0$, and $q \in \gamma_t(p).\mathcal{CN} \subseteq p.\mathcal{N}$ such that $P_{Free}(\gamma_t) = p.\mathcal{N} \backslash (\{q\} \cup q.\mathcal{N})$. Furthermore, for every execution $e' = (\gamma'_i)_{i \geq 0}$ which shares the same prefix as e between γ_0 and γ_t, $\forall t' \in \{t, ..., reqUp(e', t)\}, P_{Free}(\gamma'_{t'}) = P_{Free}(\gamma_t)$.

Using the *No Livelock* property of maximal concurrency, there also exists $t_{MC} > 0$ such that for every execution $e' = (\gamma'_i)_{i \geq 0}$, if $R(e', t, t_{MC})$ exists and $R(e', t, t_{MC}) \leq reqUp(e', t)$ then $P_{Free}(\gamma'_{R(e',t,t_{MC})}) = \emptyset$.

We build an execution e' with prefix $\gamma_0 ... \gamma_t$. Since $P_{Free}(\gamma_t) \neq \emptyset$, we are able to a add step of the algorithm from γ_t such that no request occurs and no resource is released (by *No Deadlock* property from maximal concurrency). By applying the second part of Lemma 6.2, we have $P_{Free}(\gamma'_{t+1}) = P_{Free}(\gamma_t) \neq \emptyset$. We repeat this operation until t_{MC} rounds have elapsed (this is possible since we assumed a weakly fair daemon), so that: $R(e', t, t_{MC}) \leq reqUp(e', t)$. Hence, $P_{Free}(\gamma'_{R(e',t,t_{MC})}) = P_{Free}(\gamma_t) \neq \emptyset$, contradicting the *No Livelock* property of the maximal concurrency. ■

6.5 Partial Concurrency

We now generalize the maximal concurrency to be able to define weaker degrees of concurrency that will be achievable for all instances of LRA. This generalization is called *partial concurrency*.

6.5.1 Definition

Maximal concurrency requires that a requesting process should not be prevented from accessing its critical section unless to avoid safety violations. The idea of partial concurrency is to slightly relax this property by (momentarily) blocking some requesting processes that nevertheless could enter their critical section without violating safety. We

define $\mathcal{P}$ as a predicate which represents the sets of requesting processes that can be (momentarily) blocked, while they could access their requesting resources without violating safety.

Definition 6.5 (*Partial Concurrency* w.r.t. $\mathcal{P}$)

A resource allocation algorithm ALG is *partially concurrent* w.r.t. $\mathcal{P}$ in a network $\mathcal{G} = (V, E)$ if and only if

- *No Deadlock:* For every subset of processes $X \subseteq V$, for every configuration γ, if $\mathcal{P}(X, \gamma)$ holds and $P_{Free}(\gamma) \not\subseteq X$, there exists a configuration γ' and a step $\gamma \mapsto \gamma'$ such that $ContinuousCS(\gamma\gamma') \wedge NoRequest(\gamma\gamma')$;
- *No Livelock:* There exists a number of rounds t_{PC} such that for every execution $e = (\gamma_i)_{i \geq 0}$ and for every index $i \geq 0$, if $R(e, i, t_{PC})$ exists then

$$R(e, i, t_{PC}) \leq reqUp(e, i) \Rightarrow \exists X, \mathcal{P}(X, \gamma_{R(e,i,t_{PC})}) \wedge P_{Free}(\gamma_{R(e,i,t_{PC})}) \subseteq X$$

Notice that maximal concurrency is equivalent to partial concurrency w.r.t. $\mathcal{P}_{\max}$, where $\forall X \subseteq V, \forall \gamma \in \mathcal{C}, \mathcal{P}_{\max}(X, \gamma) \equiv X = \emptyset$.

6.5.2 Strong Concurrency

The proof of Lemma 6.2 exhibits a possible scenario for some instances of LRA which shows the incompatibility of fairness and maximal concurrency: enforce maximal concurrency can lead to unfair behaviors where some neighbors of a process alternatively use resources which are conflicting with its own request. So, to achieve fairness, we must then relax the expected level of concurrency in such a way that this situation cannot occur indefinitely.

The key idea is that sometimes the algorithm should prioritize one process p against its neighbors, although it cannot immediately enter the critical section because some of its conflicting neighbors are in critical section. In this case, the algorithm should momentarily block all conflicting requesting neighbors of p that can enter critical section without violating safety, so that p enters critical section first.

In the worst case, p has only one conflicting neighbor q in critical section and so the set of processes that p has to block contains up to all conflicting (requesting) neighbors of p that are neither q, nor conflicting neighbors of q (by definition, any conflicting neighbor common to p and q cannot access critical section without violating safety because of q). We derive the following refinement of partial concurrency based on this latter observation. This property seems to be very close to the maximum degree of concurrency which can be ensured by an algorithm solving all instances of LRA.

Definition 6.6 (*Strong Concurrency*)

A resource allocation algorithm ALG is *strongly concurrent* in a network $\mathcal{G} = (V, E)$ if and only if ALG is partially concurrent w.r.t. $\mathcal{P}_{strong}$ in $\mathcal{G}$, where $\forall X \subseteq V$, $\forall \gamma \in \mathcal{C}$,

$$\mathcal{P}_{strong}(X, \gamma) \equiv \exists p \in V, \exists q \in \gamma(p).\mathcal{CN}, X = \gamma(p).\mathcal{CN} \setminus \big(\{q\} \cup \gamma(q).\mathcal{CN} \big)$$

6.6 Local Resource Allocation Algorithm

We now propose a snap-stabilizing LRA algorithm which achieves strong concurrency in identified connected networks of arbitrary topology.

6.6.1 Overview of $\mathcal{LRA}$

The overall idea of our algorithm is the following. To maximize concurrency, our algorithm should follow, as much as possible, a greedy approach: if there are requesting processes having no conflicting neighbor in the critical section, then those which have locally the highest identifier are allowed to enter critical section.

Now, the algorithm should not be completely greedy, otherwise livelock can occur at processes with low identifiers, violating the fairness of the specification.

So, the idea is to make circulating a token whose aim is to cancel the greedy approach, but only in the neighborhood of the tokenholder (the rest of the network continue to follow the greedy approach): the tokenholder, if requesting, has the priority to satisfy its request; all its conflicting neighbors are blocked until it accesses its critical section. To ensure fairness, these blockings take place even if the tokenholder cannot currently access its critical section (because maybe one of its conflicting neighbor is in critical section). Such blockings slightly degrade the concurrency, this is why our algorithm is strong, but not maximal, concurrent.

Fair Composition. Composition techniques are important in the self-stabilizing area since they allow to simplify the design, analysis, and proofs of algorithms. Consider an arbitrary composition operator $\oplus$, and two algorithms ALG_1 and ALG_2. Let e be an execution of $\text{ALG}_1 \oplus \text{ALG}_2$. Let $i \in \{1, 2\}$. We say that e is *weakly fair* w.r.t. ALG_i if there is no infinite suffix of e in which a process does not execute any action of ALG_i while being continuously enabled w.r.t. ALG_i.

Our algorithm consists of the composition of two modules: Algorithm $\mathcal{LRA}$, which manages local resource allocation, and Algorithm $\mathcal{TC}$ which provides a self-stabilizing token circulation service to $\mathcal{LRA}$, whose goal is to ensure fairness. These two modules are composed using a *fair composition* [Dol00], denoted by $\mathcal{LRA} \circ \mathcal{TC}$. In such a composition, each process executes a step of each algorithm alternately. Recall that the purpose of this composition is, in particular, to simplify the design of the algorithm: a composite algorithm written in the locally shared memory model can be translated into an equivalent non-composite algorithm.

Consider the fair composition of two algorithms ALG_1 and ALG_2. The equivalent non-composite algorithm $\text{ALG}_1 \circ \text{ALG}_2$ can be obtained by applying the following rewriting rule: In $\text{ALG}_1 \circ \text{ALG}_2$, a process has its variables in ALG_1, those in ALG_2, and an additional variable $b \in \{1, 2\}$. Assume now that ALG_1 is composed of x actions denoted by

$$\mathbf{L}_{1,i} \ :: \ G_{1,i} \ \rightarrow \ S_{1,i}, \quad \forall i \in \{1, \ldots, x\}$$

and ALG_2 is composed of y actions denoted by

$$\mathbf{L}_{2,j} \ :: \ G_{2,j} \ \rightarrow \ S_{2,j}, \quad \forall j \in \{1, \ldots, y\}$$

Then, $\text{ALG}_1 \circ \text{ALG}_2$ is composed of the following $x + y + 2$ actions:

- $\forall i \in \{1, \ldots, x\}, \quad \mathbf{L}'_{1,i} \ :: \ (b = 1) \wedge G_{1,i} \ \rightarrow \ S_{1,i}; b := 2$
- $\forall j \in \{1, \ldots, y\}, \quad \mathbf{L}'_{2,j} \ :: \ (b = 2) \wedge G_{2,j} \ \rightarrow \ S_{2,j}; b := 1$
- $\mathbf{L}_1 \ :: \ (b = 1) \wedge \bigwedge_{i=1,\ldots x} \neg G_{1,i} \wedge \bigvee_{j=1,\ldots,y} G_{2,j} \ \rightarrow \ b := 2$
- $\mathbf{L}_2 \ :: \ (b = 2) \wedge \bigwedge_{j=1,\ldots y} \neg G_{2,j} \wedge \bigvee_{i=1,\ldots,x} G_{2,i} \ \rightarrow \ b := 1$

Notice that, by definition of the composition, under the weak fair daemon assumption, no algorithm in the composition can prevent the other from executing, if this latter is continuously enabled. Rather, it can only slow down the execution by a factor 2.

Remark 6.1

Under the weakly fair daemon, in $\text{ALG}_1 \circ \text{ALG}_2$ *we have:* $\forall i \in \{1, 2\}$, $\forall p \in V$, *if* p *is continuously enabled* w.r.t. ALG_i *until (at least) executing an enabled action of* ALG_i, *then* p *executes an enabled action of* ALG_i *within at most 2 rounds.*

Remark 6.2

Under the weakly fair daemon, $\forall i \in \{1, 2\}$, *every execution of* $\text{ALG}_1 \circ \text{ALG}_2$ *is* weakly fair w.r.t. ALG_i.

Token Circulation Module. We assume that $\mathcal{TC}$ is a self-stabilizing black box which allows $\mathcal{LRA}$ to emulate a self-stabilizing token circulation. $\mathcal{TC}$ provides two outputs to each process p in $\mathcal{LRA}$: the predicate $TokenReady(p)$ and the statement $PassToken(p)$.[1]

The predicate $TokenReady(p)$ expresses the fact that the process p holds a token and can release it. Note that this interface of $\mathcal{TC}$ allows some process to hold the token without being allowed to release it yet: this may occur, for example, when, before releasing the token, the process has to wait for the network to clean some faults.

The statement $PassToken(p)$ can be used to pass the token from p to one of its neighbor. Of course, it should be executed (by $\mathcal{LRA}$) only if $TokenReady(p)$ holds.

Precisely, we assume that $\mathcal{TC}$ satisfies the three following properties.

[1]Since $\mathcal{TC}$ is a black box with only two outputs, $TokenReady(p)$ and $PassToken(p)$, these outputs are the only part of $\mathcal{TC}$ that $\mathcal{LRA}$ can use.

Property 6.1 (*Stabilization*)

Consider an arbitrary composition of $\mathcal{TC}$ and some other algorithm. Let e be any execution of this composition which is weakly fair w.r.t. *$\mathcal{TC}$.*

If for any process p, $PassToken(p)$ is executed in e only when $TokenReady(p)$ holds, then $\mathcal{TC}$ stabilizes in e, i.e., reaches and remains in configurations where there is a unique token in the network, independently of any call to $PassToken(p)$ at any process p.

In other words, Property 6.1 means that, even if $PassToken$ is never called, $\mathcal{TC}$ stabilizes.

Property 6.2 (*Token Consistency*)

Consider an arbitrary composition of $\mathcal{TC}$ and some other algorithm. Let e be any execution of this composition which is weakly fair w.r.t. *$\mathcal{TC}$ and where $\mathcal{TC}$ is stabilized.*

Then, $\forall p \in V$, each time $TokenReady(p)$ holds in e, $TokenReady(p)$ is continuously true in e until $PassToken(p)$ is invoked.

Property 6.3 (*Fairness*)

Consider an arbitrary composition of $\mathcal{TC}$ and some other algorithm. Let e be any execution of this composition which is weakly fair w.r.t. *$\mathcal{TC}$ and where $\mathcal{TC}$ is stabilized.*

If $\forall p \in V$,

- *$PassToken(p)$ is invoked in e only when $TokenReady(p)$ holds, and*
- *if $PassToken(p)$ is invoked within finite time in e each time $TokenReady(p)$ holds,*

then $\forall p \in V$, $TokenReady(p)$ holds infinitely often in e.

To design $\mathcal{TC}$, we proceed as follows. There exist several self-stabilizing token circulations for arbitrary rooted networks [CDV09, DJPV00, HC93] that contain a particular action,

$$T : TokenReady(p) \rightarrow PassToken(p)$$

to pass the token, and that stabilizes independently of the activations of action T.

Now, the networks we consider are not rooted, but identified. So, to obtain a self-stabilizing token circulation for arbitrary identified networks, we can fairly compose any of them with a self-stabilizing leader election algorithm, *e.g.*, [AG94, DH97, DLV11a, DLV11b] or [ACD+16] (see Chapter 4 for a more detailled state of the art) using the following additional rule: if a process considers itself as leader it executes the token circulation local algorithm for a root; otherwise it executes the local algorithm for a non-root. Finally, we obtain $\mathcal{TC}$ by removing action T from the resulting algorithm, while keeping $TokenReady(p)$ and $PassToken(p)$ as outputs, for every process p.

Remark 6.3

Following Properties 6.2 and 6.3, the algorithm, noted $\mathcal{TC}^*$, made of Algorithm $\mathcal{TC}$ where action $T : TokenReady(p) \rightarrow PassToken(p)$ has been added, is a self-stabilizing token circulation.

The algorithm presented in next section for local resource allocation emulates action T using predicate $TokenReady(p)$ and statement $PassToken(p)$ given as inputs.

Resource Allocation Module. The code of $\mathcal{LRA}$ is given in Algorithm 13. Priorities and guards ensure that actions of Algorithm 13 are mutually exclusive. We now informally describe Algorithm 13, and explain how the properties of the specification (Definition 6.1, page 174) is instantiated with its variables.

First, a process p interacts with its application through two variables: $p.req \in \mathcal{R}_p \cup \{\perp\}$ and $p.status \in \{\text{Out}, \text{Wait}, \text{In}, \text{Blocked}\}$. $p.req$ is an input that can be read and written by the application, but can only be read by p in $\mathcal{LRA}$. Conversely, $p.status$ can be read and written by p in $\mathcal{LRA}$, but the application can only read it.

Variable $p.status$ can take the following values:

- Wait, which means that p requests a resource but does not hold it yet;
- Blocked, which means that p requests a resource, but cannot hold it now;
- In, which means that p holds a resource;
- Out, which means that p is currently not involved into an allocation process.

When $p.req = \perp$, this means that no resource is requested. Conversely, when $p.req \in \mathcal{R}_p$, the value of $p.req$ informs p about the resource the application requests. We assume two properties on $p.req$. Property 6.4 ensures that the application:

1. does not request for resource r' while a computation to access resource r is running
2. does not cancel or modify a request before the request is satisfied.

Property 6.5 ensures that any critical section is finite.

Property 6.4

$\forall p \in V$, the updates on $p.req$ (by the application) satisfy the following constraints:

- *The value of $p.req$ can be switched from $\perp$ to $r \in \mathcal{R}_p$ if and only if $p.status =$ Out,*
- *The value of $p.req$ can be switched from $r \in \mathcal{R}_p$ to $\perp$ (meaning that the application leaves the critical section) if and only if $p.status =$ In.*
- *The value of $p.req$ cannot be directly switched from $r \in \mathcal{R}_p$ to $r' \in \mathcal{R}_p$ with $r' \neq r$.*

Property 6.5

$\forall p \in V$, if $p.status =$ In and $p.req \neq \perp$, then eventually $p.req$ becomes $\perp$.

Algorithm 13 – Actions of Process p in Algorithm $\mathcal{LRA}$.

Inputs.

- $p.id \in \mathfrak{id}$
- $p.\mathcal{N}$
- $p.req \in \mathcal{R}_p \cup \{\bot\}$
- $TokenReady(p)$, predicate from $\mathcal{TC}$
- $PassToken(p)$, statement from $\mathcal{TC}$

Variables.

- $p.status \in \{\text{Out}, \text{Wait}, \text{Blocked}, \text{In}\}$
- $p.token \in \mathbb{B}$

Functions.

$$
\begin{array}{lcl}
Candidates(p) & \equiv & \{q \in p.\mathcal{N} \cup \{p\} : q.status = \text{Wait}\} \\
TokenCand(p) & \equiv & \{q \in Candidates(p) : q.token\} \\
Winner(p) & \equiv & \begin{cases} \max\{q \in TokenCand(p)\} & \text{if } TokenCand(p) \neq \emptyset, \\ \max\{q \in Candidates(p)\} & \text{otherwise.} \end{cases}
\end{array}
$$

Predicates.

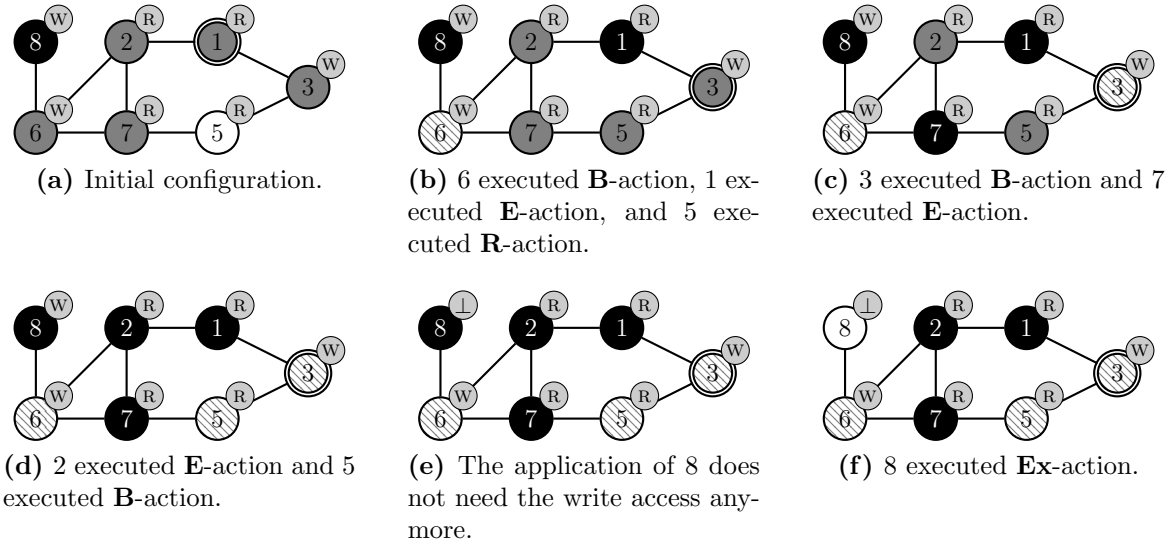
$$
\begin{array}{lcl}
RsrcFree(p) & \equiv & \forall q \in p.\mathcal{N}, (q.status = \text{In} \Rightarrow p.req \rightleftharpoons q.req) \\
IsBlocked(p) & \equiv & \neg RsrcFree(p) \vee (\exists q \in p.\mathcal{N}, q.status = \text{Blocked} \wedge q.token \\
 & & \quad \wedge\ p.req \neq q.req)
\end{array}
$$

Guards.

$$
\begin{array}{lcl}
Request(p) & \equiv & p.status = \text{Out} \wedge p.req \neq \bot \\
Block(p) & \equiv & p.status = \text{Wait} \wedge IsBlocked(p) \\
Unblock(p) & \equiv & p.status = \text{Blocked} \wedge \neg IsBlocked(p) \\
Enter(p) & \equiv & p.status = \text{Wait} \wedge \neg IsBlocked(p) \wedge p = Winner(p) \\
Exit(p) & \equiv & p.status \neq \text{Out} \wedge p.req = \bot \\
ResetToken(p) & \equiv & TokenReady(p) \neq p.token \\
ReleaseToken(p) & \equiv & TokenReady(p) \wedge p.status \in \{\text{Out}, \text{In}\} \wedge \neg Request(p)
\end{array}
$$

Actions.

RsT	(prio. 1)	::	$ResetToken(p)$	$\rightarrow$	$p.token := TokenReady(p)$
Ex	(prio. 2)	::	$Exit(p)$	$\rightarrow$	$p.status := \text{Out}$
RlT	(prio. 3)	::	$ReleaseToken(p)$	$\rightarrow$	$PassToken(p)$
R	(prio. 3)	::	$Request(p)$	$\rightarrow$	$p.status := \text{Wait}$
B	(prio. 3)	::	$Block(p)$	$\rightarrow$	$p.status := \text{Blocked}$
U	(prio. 3)	::	$Unblock(p)$	$\rightarrow$	$p.status := \text{Wait}$
E	(prio. 3)	::	$Enter(p)$	$\rightarrow$	$p.status := \text{In}$ **if** $TokenReady(p)$ **then** $PassToken(p)$



(a) Initial configuration.

(b) 6 executed **B**-action, 1 executed **E**-action, and 5 executed **R**-action.

(c) 3 executed **B**-action and 7 executed **E**-action.

(d) 2 executed **E**-action and 5 executed **B**-action.

(e) The application of 8 does not need the write access anymore.

(f) 8 executed **Ex**-action.

Figure 6.4 – Example of execution of $\mathcal{LRA} \circ \mathcal{TC}$. The status of a process is represented by the color of the corresponding node: white for Out, gray for Wait, black for In, hatched for Blocked. Double circled nodes hold a token. The requesting resource is inside the bubble next to the node.

Then, we instantiate the predicates used by the specification in Definition 6.1. The predicate $Req(\gamma_i, p, r)$ is given by $Req(\gamma_i, p, r) \equiv \gamma_i(p).req = r$.

Remind that $\perp$ compatible with every resource. The predicate $NoConflict(\gamma_i, p)$ is expressed by $NoConflict(\gamma_i, p) \equiv \gamma_i(p).status = \text{In} \Rightarrow \big(\forall q \in p.\mathcal{N}, \gamma_i(q).status = \text{In} \Rightarrow (\gamma_i(q).req \rightleftharpoons \gamma_i(p).req)\big)$.

The predicate $Start(\gamma_i, \gamma_{i+1}, p, r)$ becomes true when process p takes the request for resource r into account in $\gamma_i \mapsto \gamma_{i+1}$, *i.e.*, when the status of p switches from Out to Wait in $\gamma_i \mapsto \gamma_{i+1}$ because $p.req = r \neq \perp$ in γ_i:

$$\begin{aligned} Start(\gamma_i, \gamma_{i+1}, p, r) \quad \equiv \quad & \gamma_i(p).status = \text{Out} \wedge \gamma_{i+1}(p).status = \text{Wait} \\ & \wedge\ \gamma_i(p).req = \gamma_{i+1}(p).req = r \end{aligned}$$

A computation $\gamma_i \dots \gamma_j$ where $Result(\gamma_i \dots \gamma_j, p, r)$ holds means that p accesses resource r, *i.e.*, p switches its status from Wait to In in $\gamma_{i-1} \mapsto \gamma_i$ while $p.req = r$, and later switches its status from In to Out in $\gamma_j \mapsto \gamma_{j+1}$. So,

$$\begin{aligned} Result(\gamma_i \dots \gamma_j, p, r) \quad \equiv \quad & \gamma_i(p).status = \text{Wait} \wedge \gamma_i(p).req = \gamma_{i+1}(p).req = r \\ & \wedge\ \forall k \in \{i+1, \dots, j-1\}, \big(\gamma_k(p).status = \text{In} \\ & \wedge\ \gamma_j(p).status = \text{Out} \wedge \gamma_j(p).req = \perp\big) \end{aligned}$$

We now illustrate the principles of $\mathcal{LRA}$ with the example given in Figure 6.4. In this example, we consider the local readers-writers problem. Recall that we have two resources: R for a reading access and W for a writing access, with $\text{R} \rightleftharpoons \text{R}$, $\text{R} \not\rightleftharpoons \text{W}$ and $\text{W} \not\rightleftharpoons \text{W}$.

When the process is idle ($p.status = \text{Out}$), its application can request a resource. In this case, $p.req \neq \perp$ and p sets $p.status$ to Wait by **R**-action: p starts the computation to obtain the resource. For example, 5 starts a computation to obtain r in (a)↦(b). If one of its neighbors is using a conflicting resource, p cannot satisfy its request yet. So, p switches $p.status$ from Wait to Blocked by **B**-action (see 6 in (a)↦(b)). If there is no more neighbor using conflicting resources, p gets back to status Wait by **U**-action.

When several neighbors request for conflicting resources, we break ties using a token-based priority: Each process p has an additional Boolean variable $p.token$ which is used to inform neighbors about whether p holds a token or not. A process p takes priority over any neighbor q if and only if $\big(p.token \wedge \neg q.token\big) \vee \big(p.token = q.token \wedge p > q\big)$[2]. More precisely, if there is no waiting tokenholder in the neighborhood of p, the highest priority process is the waiting process with highest ID. This highest priority process is $Winner(p)$. Otherwise, the tokenholders (there may be several tokens during the stabilization phase of $\mathcal{TC}$) block all their requesting neighbors, except the ones requesting for non-conflicting resources until they obtain their requested resources. This mechanism allows to ensure fairness by slightly decreasing the level of concurrency. (The token circulates to eventually give priority to blocked processes, *e.g.*, processes with small IDs.)

The highest priority waiting process in the neighborhood gets status In and can use its requested resource by **E**-action, *e.g.*, 7 in step (b)↦(c) or 1 in (a)↦(b). Moreover, if it holds a token, a tokenholder releases it when accessing its requested resource. Notice that, as a process is not blocked when one of its neighbors is requesting/using a compatible resource, several neighbors requesting/using compatible resources can concurrently enter/execute their critical section (see 1, 2, and 7 in Configuration (d)). When the application at process p does not need the resource anymore, *i.e.*, when it sets the value of $p.req$ to $\perp$. Then, p executes **Ex**-action and switches its status to Out, *e.g.*, 8 during step (e)↦(f).

RlT-action is used to straight away pass the token to a neighbor when the process does not need it, *i.e.*, when either its status is Out and no resource is requested or when its status is In. (Hence, the token can eventually reach a requesting process and help it to satisfy its request.)

The last action, **RsT**-action, ensures the consistency of variable *token* so that the neighbors realize whether or not a process holds a token. Indeed, the additional variable $p.token$ is necessary when the predicate $TokenReady(p)$ involve variables of some neighbors of p.

Hence, any request is satisfied in a finite time. As an illustrative example, consider the local mutual exclusion problem and the execution given in Figure 6.5. In this example, we try to delay as much as possible the critical section of process 2. First, process 2 has two neighbors (7 and 8) that also request the resource and have greater IDs. So, they will execute their critical section before 2 (in steps (a)↦(b) and (e)↦(f)). But, the token circulates and eventually reaches 2 (see Configuration (g)). Then, 2 has priority over its neighbors (even though it has a lower ID) and eventually starts executing its critical

[2]Notice that when two neighbors simultaneously hold the token (only during the stabilization phase of $\mathcal{TC}$), the one with the highest identifier has priority.

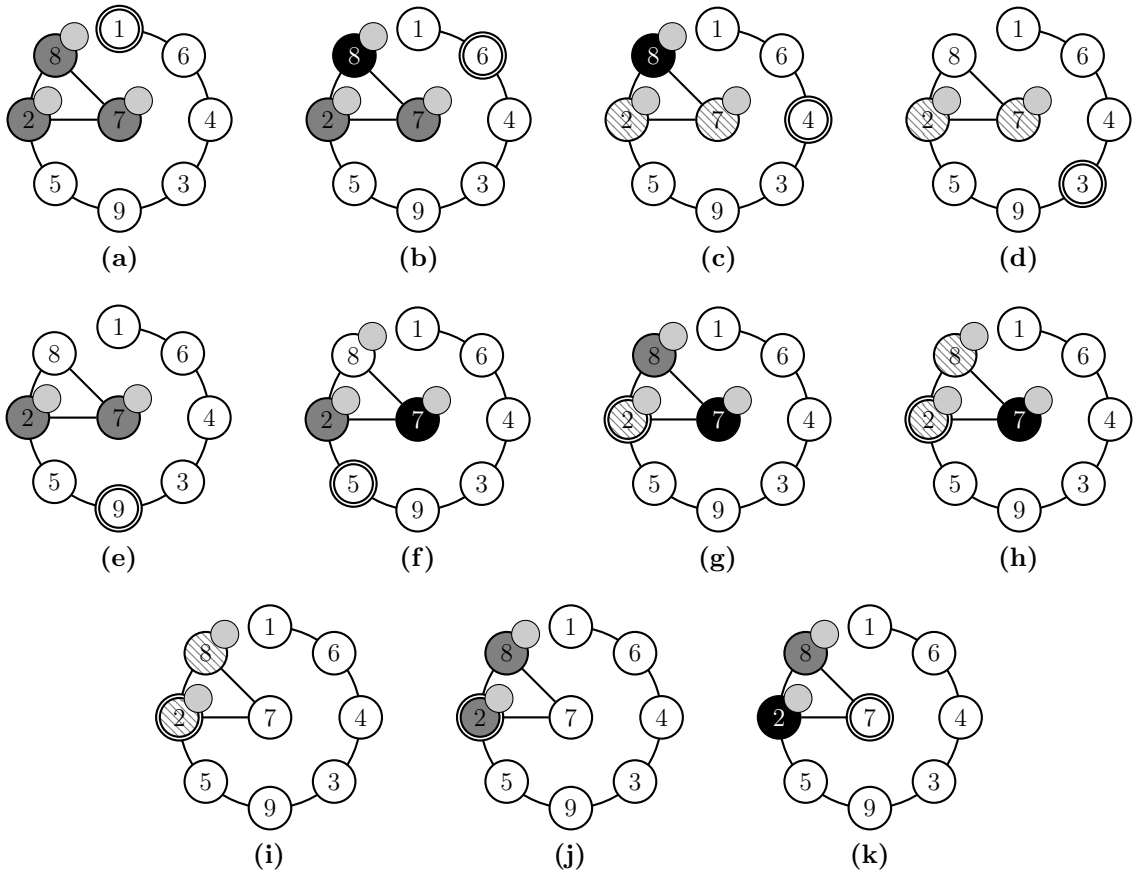


Figure 6.5 – Example of execution of $\mathcal{LRA} \circ \mathcal{TC}$ on the local mutual exclusion problem. The bubbles mark the requesting processes.

section in (j)↦(k).

6.6.2 Correctness and Complexity Analysis of $\mathcal{LRA} \circ \mathcal{TC}$

In this section, we prove the correctness and we study the complexity of $\mathcal{LRA} \circ \mathcal{TC}$.

Correctness. In this subsection, we prove that $\mathcal{LRA} \circ \mathcal{TC}$ is snap-stabilizing w.r.t. SP_{LRA} (see Definition 6.1, page 174), assuming a distributed weakly fair daemon. First, we show the safety part, namely, the *Resource Conflict Freedom* property is always satisfied. Then, we assume a distributed weakly fair daemon to prove the liveness part, *i.e.*, the *Computation Start* and *Computation End* properties.

Remark 6.4

If **E***-action is enabled at a process* p *in a configuration* γ*, then* $\forall q \in p.\mathcal{N}$*,*

$$\big(\gamma(q).status = \text{In} \Rightarrow \gamma(p).req \rightleftharpoons \gamma(q).req\big)$$

Lemma 6.3

__E__-action cannot be simultaneously enabled at two neighbors.

Proof: Let γ be a configuration. Let $p \in V$ and $q \in p.\mathcal{N}$. Assume by contradiction that **E**-action is enabled at p and q in γ. Then, $\gamma(p).status = \gamma(q).status = \text{WAIT}$ and both $p = Winner(p)$ and $q = Winner(q)$ hold in γ. Since by definition, $p, q \in Candidates(p)$ and $p, q \in Candidates(q)$, we obtain a contradiction. ■

Lemma 6.4

Let $\gamma \mapsto \gamma'$ be a step. Let $p \in V$. If $NoConflict(\gamma, p)$ holds, then $NoConflict(\gamma', p)$ holds.

Proof: Let $\gamma \mapsto \gamma'$ be a step. Let $p \in V$. Assume by contradiction that $NoConflict(\gamma, p)$ holds but $\neg NoConflict(\gamma', p)$. Then, $\gamma'(p).status = \text{IN}$ and $\exists q \in p.\mathcal{N}$ such that $\gamma'(q).status = \text{IN}$ and $\gamma'(q).req \neq \gamma'(p).req$. As a consequence, $\gamma'(p).req \in \mathcal{R}_p$ and $\gamma'(p).req \in \mathcal{R}_q$.

Using Property 6.4,

- The value of $p.req$ can be switched from $\perp$ in γ to $r \in \mathcal{R}_p$ in γ' only if $\gamma(p).status = \text{OUT}$. But $\gamma'(p).status = \text{IN}$ and it is impossible to switch $p.status$ from OUT to IN in one step.
- The value of $p.req$ cannot be switched from $r' \in \mathcal{R}_p$ in γ to $r \in \mathcal{R}_p$ with $r \neq r'$.

Hence, $\gamma(p).req = \gamma'(p).req \in \mathcal{R}_p$. We can make the same reasoning on q so $\gamma(q).req = \gamma'(q).req \in \mathcal{R}_q$, and $\gamma(q).req \neq \gamma(p).req$. Now, there are two cases:

1. If $\gamma(p).status = \text{IN}$, as $NoConflict(\gamma, p)$ holds, $\forall x \in p.\mathcal{N}, (\gamma(x).status = \text{IN} \Rightarrow \gamma(p).req \rightleftharpoons \gamma(x).req)$. In particular, $\gamma(q).status \neq \text{IN}$, since $\gamma(q).req \neq \gamma(p).req$. So q executes **E**-action $\gamma \mapsto \gamma'$ to obtain status IN. This contradicts Remark 6.4, since q has a conflicting neighbor (p) with status IN in γ.
2. If $\gamma(p).status \neq \text{IN}$, then p executes **E**-action in step $\gamma \mapsto \gamma'$ to get status IN. Now, there are two cases:
   a) If $\gamma(q).status \neq \text{IN}$, then q executes **E**-action in $\gamma \mapsto \gamma'$. So **E**-action is enabled at p and q in γ, a contradiction to Lemma 6.3.
   b) If $\gamma(q).status = \text{IN}$, then **E**-action is enabled at p in γ although a neighbor of p has status IN and a conflicting request (p is in a similar situation to the one of q in case 1), a contradiction to Remark 6.4. ■

Theorem 6.2 (*Resource Conflict Freedom*)

Any execution of $\mathcal{LRA} \circ \mathcal{TC}$ satisfies the resource conflict freedom property.

Proof: Let $e = (\gamma_i)_{i \geq 0}$ be an execution $\mathcal{LRA} \circ \mathcal{TC}$. Let $k \geq 0$ and $k' > k$. Let $p \in V$. Let $r \in \mathcal{R}_p$. Assume $Result(\gamma_k \ldots \gamma_{k'}, p, r)$. Assume $\exists l < k$ such that $Start(\gamma_l, \gamma_{l+1}, p, r)$. In particular, $\gamma_l(p).status \neq \text{IN}$. Hence, $NoConflict(\gamma_l, p)$ trivially holds. Using Lemma 6.4, $\forall i \geq l$, $NoConflict(\gamma_i, p)$ holds. In particular, $\forall i \in \{k, \ldots, k'\}$, $NoConflict(\gamma_i, p)$. ■

In the following, we assume a weakly fair daemon.

Lemma 6.5

The stabilization of $\mathcal{TC}$ is preserved by fair composition.

Proof: By definition of the algorithm, for any process p, $PassToken(p)$ is executed only in $\mathcal{LRA}$ when $TokenReady(p)$ holds (see **RlT**-action and **E**-action). Moreover, by Remark 6.2, every execution of $\mathcal{LRA} \circ \mathcal{TC}$ is *weakly fair* w.r.t. $\mathcal{TC}$. So, $\mathcal{TC}$ self-stabilizes to a unique tokenholder in every execution of $\mathcal{LRA} \circ \mathcal{TC}$, by Property 6.1. ■

Lemma 6.6

A process cannot keep a token forever in $\mathcal{LRA} \circ \mathcal{TC}$.

Proof: Let e be an execution. By Lemma 6.5, the token circulation eventually stabilizes, *i.e.*, there is a unique token in every configuration after stabilization of $\mathcal{TC}$. Assume by contradiction that, after such a configuration γ, a process p keeps the token forever: $TokenReady(p)$ holds forever and $\forall q \in V$ with $q \neq p$, $\neg TokenReady(q)$ holds forever.

First, the values of *token* variables are eventually updated to the corresponding value of the predicate $TokenReady$. Indeed, the values of predicate $TokenReady$ do not change anymore. So, if there is $x \in V$ such that $x.token \neq TokenReady(x)$, **RsT**-action (the highest priority action of $\mathcal{LRA}$) is continuously enabled at x, until x executes it. Now, by Remark 6.1, in finite time, x executes **RsT**-action to update its *token* variable. Therefore, in finite time, the system reaches and remains in configurations where $p.token = \text{TRUE}$ forever and $\forall q \in V$ with $q \neq p$, $q.token = \text{FALSE}$ forever. Let γ' be such a configuration. Notice that **RsT**-action is continuously disabled from γ'. Then, we can distinguish six cases:

1. If $\gamma'(p).status = \text{WAIT}$ and $\gamma'(p).req \neq \perp$, then $TokenCand(p) = \{p\}$ and so $Winner(p) = p$ holds forever, and $\forall q \in p.\mathcal{N}$, $TokenCand(q) = \{p\}$ and $Winner(q) = p \neq q$ holds forever. **E**-action is disabled forever at q from γ'. Now, if $\exists q \in p.\mathcal{N}$ such that $\gamma'(q).status = \text{IN} \wedge \gamma'(q).req \neq \gamma'(p).req$, then, as $\perp$ is compatible with any resource, $\gamma'(q).req \neq \perp$. Using Property 6.5, in finite time the request of q becomes $\perp$ and remains $\perp$ until q obtains status OUT (Property 6.4). So **Ex**-action is continuously enabled at q, until q executes it. Hence, by Remark 6.1, in finite time, those processes leave critical section and cannot enter again since **E**-action is disabled forever, and so $\forall q \in p.\mathcal{N}$, $q.status \neq \text{IN}$ forever. So $IsBlocked(p)$ does not hold anymore. Notice that, if p gets status BLOCKED in the meantime, **U**-action is continuously enabled at p until p executes it, so p gets back status WAIT in finite time by Remark 6.1. Then, $Winner(p) = p$ still holds so **E**-action is continuously enabled at p, until p executes it. Hence, by Remark 6.1, in finite time, p executes **E**-action and releases its token, a contradiction.
2. If $\gamma'(p).status = \text{OUT}$ and $\gamma'(p).req \neq \perp$, the application cannot modify $p.req$ until p enters its critical section (Property 6.4). Hence, **RlT**-action is disabled until p gets status IN. So, **R**-action is continuously enabled at p until p executes it, and p eventually gets status WAIT by Remark 6.1. We then reach case 1 and we are done.
3. If $\gamma'(p).status = \text{OUT}$ and $\gamma'(p).req = \perp$. If eventually $p.req \neq \perp$, then we retrieve case 2, a contradiction. Otherwise, **RlT**-action is continuously enabled at p until p

executes it. So, by Remark 6.1, in finite time, p executes **RlT**-action and releases its token by a call to $PassToken(p)$, a contradiction.

4. If $\gamma'(p).status = \text{BLOCKED}$ and $\gamma'(p).req \neq \bot$, then $p.status = \text{BLOCKED}$ forever from γ', otherwise we eventually retrieve case 1. So, $\forall q \in p.\mathcal{N}$ such that $\gamma'(p).req \neq \gamma'(q).req$, $IsBlocked(q)$ holds forever so **E**-action is disabled at q forever. Now, as in case 1, $\forall q \in p.\mathcal{N}$ such that $\gamma'(p).req \neq \gamma'(q).req$, we have $q.status \neq \text{IN}$ forever after a finite time. So, eventually **U**-action is continuously enabled at p until p executes it. Hence, by Remark 6.1, in finite time, p gets status WAIT and we retrieve case 1, a contradiction.
5. If $\gamma'(p).status \in \{\text{WAIT}, \text{BLOCKED}\}$ and $\gamma'(p).req = \bot$. If eventually $p.req \neq \bot$, then we retrieve cases 1 or 4, a contradiction. Otherwise, **Ex**-action is continuously enabled at p until p executes it. So, by Remark 6.1, in finite time, p executes **Ex**-action and we retrieve case 3, a contradiction.
6. If $\gamma'(p).status = \text{IN}$, either $\gamma'(p).req = \bot$ or in finite time $p.req$ becomes $\bot$ (Property 6.5) and remains $\bot$ until p obtains status OUT (Property 6.4). Once $p.req = \bot$, **Ex**-action is continuously enabled at p until p executes it. So, by Remark 6.1 p eventually gets status OUT, and we retrieve case 3, a contradiction. ■

Lemma 6.6 implies that the hypothesis of Property 6.3 is satisfied. Hence, we can deduce Corollary 6.1.

Corollary 6.1

After stabilization of the token circulation module, $TokenReady(p)$ holds infinitely often at any process p in $\mathcal{LRA} \circ \mathcal{TC}$.

Lemma 6.7

If $Exit(p)$ continuously holds at some process p until it executes **Ex***-action, then p executes* **Ex***-action in finite time.*

Proof: Assume, by the contradiction, that from some configuration $Exit(p)$ continuously holds, but p never executes **Ex**-action. Then, remind that **RlT**-action and **E**-action are the only actions allowing p to release a token. Now, $Exit(p)$ is the guard of action **Ex**-action whose priority is higher than those of **RlT**-action and **E**-action. So, p never more releases a token, a contradiction to Lemma 6.5 and Corollary 6.1. ■

Lemma 6.8

If $Request(p)$ continuously holds at some process p until it executes **R***-action, then p executes* **R***-action in finite time.*

Proof: Assume, by the contradiction, that from some configuration $Request(p)$ continuously holds, but p never executes **R**-action. Then, remind that **RlT**-action and **E**-action are the only actions allowing p to release a token. Now, $Request(p)$ implies $\neg ReleaseToken(p)$, so **RlT**-action is disabled at p forever. Moreover, $Request(p)$ is the guard of action **R**-action whose priority is higher than the one of **E**-action. So, p never more releases a token, a contradiction to Lemma 6.5 and Corollary 6.1. ■

Lemma 6.9

Any process p such that $p.status \in \{\text{WAIT}, \text{BLOCKED}\}$ and $p.req \neq \perp$ executes **E***-action in finite time.*

Proof: Let e be an execution, $\gamma \in e$ be a configuration, and $p \in V$ such that $\gamma(p).status \in \{\text{WAIT}, \text{BLOCKED}\}$ and $\gamma(p).req \neq \perp$. Then, $\gamma(p).req \neq \perp$ holds while $\gamma(p).status \neq \text{IN}$ (Property 6.4). So, while p does not execute **E**-action, $p.status \in \{\text{WAIT}, \text{BLOCKED}\}$ and $p.req \neq \perp$. Now, by Lemma 6.5, the token circulation eventually stabilizes. By Corollary 6.1, in finite time p holds the unique token. From this configuration, p cannot keep forever the token (Lemma 6.6) and p can only release it by executing **E**-action (by Property 6.2). ■

Lemma 6.10

Any process of status different from OUT *sets its variable status to* OUT *within finite time.*

Proof: Let $p \in V$. Let γ be a configuration. Assume first $\gamma(p).status = \text{IN}$. If $\gamma(p).req \neq \perp$, in finite time $p.req$ is set to $\perp$ (Property 6.5) and then cannot be modified until p gets status OUT (Property 6.4). So, $Exit(p)$ continuously holds until p executes **Ex**-action. Then, **Ex**-action is executed by p in finite time, by Lemma 6.7: p gets status OUT.

Assume now that $\gamma(p).status \in \{\text{WAIT}, \text{BLOCKED}\}$. If eventually $p.req \neq \perp$, then p executes **E**-action in a finite time (Lemma 6.9). So, p eventually gets status IN and we retrieve the previous case. Otherwise, $Exit(p)$ continuously holds until p executes **Ex**-action. Then, **Ex**-action is executed by p in finite time, by Lemma 6.7: p gets status OUT. ■

Notice that if a process that had status WAIT or BLOCKED obtains status OUT, this means that its computation ended.

Theorem 6.3 (*Computation Start*)

Any execution of $\mathcal{LRA} \circ \mathcal{TC}$ satisfies the Computation Start property.

Proof: Let $e = (\gamma_i)_{i \geq 0}$ be an execution. Let $k \geq 0$. Let $p \in V$. Let $r \in \mathcal{R}_p$. First, p eventually has status OUT, by Lemma 6.10, let say in $\gamma_{j-1} \mapsto \gamma_j$ $(j \geq k)$. Now, if $\gamma_j(p).req \neq \perp$ holds, it holds continuously while $p.status = \text{OUT}$ (Property 6.4). So, $Request(p)$ continuously holds until p executes **R**-action. By Lemma 6.8, p eventually executes **R**-action, let say in $\gamma_l \mapsto \gamma_{l+1}$, $l \geq j \geq k$. Then, $\gamma_{l+1}(p).status = \text{WAIT}$. Notice that the application of p cannot modify its request (Property 6.4), so $\gamma_l(p).req = \gamma_{l+1}.req = r$. Hence, $Req(\gamma_l, p, r)$ and $Start(\gamma_l, \gamma_{l+1}, p, r)$ hold. ■

Theorem 6.4 (*Computation End*)

Any execution of $\mathcal{LRA} \circ \mathcal{TC}$ satisfies the Computation End property.

Proof: Let $e = (\gamma_i)_{i \geq 0}$ be an execution. Let $k \geq 0$. Let $p \in V$. Let $r \in \mathcal{R}_p$. If $Start(\gamma_k, \gamma_{k+1}, p, r)$ holds, then $\gamma_{k+1}(p).status = \text{WAIT}$ and $\gamma_{k+1}(p).req = r$. Using

Lemma 6.9, in finite time, p executes **E**-action and gets status In (let say in $\gamma_{l-1} \mapsto \gamma_l$, $l > k$). Notice that the application cannot modify the value of req until p obtains status In (Property 6.4) so $\gamma_{l-1}(p).req = \gamma_l(p).req = \gamma_{k+1}(p).req = r$. By Property 6.5 and from the algorithm, $p.status$ = In while $p.req \neq \bot$ and the application sets within finite time $p.req$ to $\bot$ (this is the only modification that can be made on $p.req$). Then, $p.req = \bot$ until $p.status$ = Out, still by Property 6.5, and from the algorithm, p can only switch $p.status$ from In to Out. So, $p.req = \bot$ continuously and $Exit(p)$ continuously holds until p executes **Ex**-action. Then, by Lemma 6.7: there is a step $\gamma_{l'} \mapsto \gamma_{l'+1}$ (with $l' \geq l$), where p executes **Ex**-action to switch $p.status$ from In to Out. So, $\gamma_{l'}(p).status$ = In and $\gamma_{l'+1}(p).status$ = Out. Consequently, $Result(\gamma_l \dots \gamma_{l'}, p, r)$ holds. ■

Using Theorems 6.2, 6.3, and 6.4, we can conclude:

Theorem 6.5 (*Correctness*)

Algorithm $\mathcal{LRA} \circ \mathcal{TC}$ is snap-stabilizing w.r.t. *SP_{LRA} assuming a distributed weakly fair daemon.*

Complexity Analysis. In this subsection, we analyze the waiting time, *i.e.*, the number of rounds required to obtain critical section after a request. Here, we assume that the execution of critical section *lasts at most C rounds*.

Lemma 6.11

In $\mathcal{LRA} \circ \mathcal{TC}$, after stabilization of $\mathcal{TC}$, there are at most $2C + 14$ rounds between a step where $TokenReady(p)$ becomes True *and the execution of $PassToken(p)$.*

Proof: As $PassToken$ is only executed in the $\mathcal{LRA}$ part of $\mathcal{LRA} \circ \mathcal{TC}$, we focus on counting rounds from $\mathcal{LRA}$, first. Then, the result has to be multiply by 2 due to the composition with $\mathcal{TC}$ (Remark 6.1).

Let p be a process. After stabilization of the token circulation algorithm, a process can only release its token by executing either **RlT**-action or **E**-action (Property 6.2).

Assume $TokenReady(p)$ holds. In one round, the variables *token* are correctly evaluated thanks to **RsT**-action executions (remind that **RsT**-action is the highest priority action). Then, there are three cases:

1. Assume p is requesting but does not get the critical section yet. In the worst case, $p.status$ = Out and $p.req \neq \bot$. In one round, p executes **R**-action and gets status Wait. Then, if there are some neighbors of p in critical section that are using a conflicting resource, they end their critical section (*i.e.*, their variable req becomes $\bot$) within the C next rounds and p executes **B**-action during the first of these C rounds. Notice that, as p holds the unique token and the *token* variables are correctly evaluated, no other neighbor of p can enter the critical section meanwhile. In the worst case, every neighbor is out of the critical section (*i.e.*, their variable req becomes $\bot$, which is compatible with any other resource) after these C rounds. Finally, p is no more blocked and executes **U**-action in one round before executing **E**-action within another round. Executing **E**-action, p releases its token. Hence, overall p releases its token within $C + 4$ rounds in this case.

2. Assume $p.req = \perp$. If $p.req$ becomes different from $\perp$ within one round, then this means that $p.status = \text{Out}$ (Property 6.4) and we retrieve the previous case and overall p releases its token within $C + 5$ rounds. Otherwise, p satisfies $p.status = \text{Out}$ in one round, by **Ex**-action if necessary. Again, either $p.req$ becomes different from $\perp$ within the next round, we retrieve the previous case, and overall p releases its token within $C + 6$ rounds, or p executes **RlT**-action during the round. So, in this latter case, p releases its token within 3 rounds.

3. Assume $p.status = \text{In}$ and $p.req \neq \perp$. If $p.req$ becomes $\perp$ within one round, we retrieve the case 2. So overall p releases its token within $C + 7$ rounds. Otherwise, **RlT**-action is continuously enabled and executed by p within the round. So, p releases its token within two rounds in this latter case.

■

Let T_S be the stabilization time in rounds of $\mathcal{TC}$. Let T_{tok} be a bound on the number of rounds required to obtain the unique token in $\mathcal{TC}$ (the algorithm obtained when adding action $T : TokenReady(p) \to PassToken(p)$ to $\mathcal{TC}$, see Remark 6.3, page 188) after its stabilization. Let N_{tok} be a bound on the number of $PassToken$ realized between two consecutive executions of $PassToken$ at the same process.

Theorem 6.6 (*Waiting Time*)

A requesting process obtains access to critical section in at most $2(T_s + T_{tok}) + (2C + 14) \times (N_{tok} + 1)$ *rounds.*

Proof: Let $p \in V$ such that $p.req \neq \perp$ and $p.status \neq \text{In}$. In the worst case, p must wait to hold a token and be the unique tokenholder to get its critical section. $\mathcal{TC}$ stabilizes in $2T_S$ rounds (the factor 2 comes from the composition, see Remark 6.1). Then, in at most $2T_{tok} + (2C + 14) \times N_{tok}$, p gets the token, since it has to wait $2T_{tok}$ rounds due to Algorithm $\mathcal{TC}$ (again, the factor 2 comes from the composition, see Remark 6.1) and $(2C + 14) \times N_{tok}$ rounds due to Algorithm $\mathcal{LRA}$. Indeed, while executing action $T :: TokenReady \to PassToken$ is atomic in $\mathcal{TC}$, a process keeps the token at most $2C + 14$ rounds in $\mathcal{LRA} \circ \mathcal{TC}$ (Lemma 6.11). Finally, to obtain critical section, it is required that p executes **E**-action which also releases the token: by Lemma 6.11 again, this may require $2C + 14$ additional rounds. Hence, in at most $2(T_s + T_{tok}) + (2C + 14) \times (N_{tok} + 1)$ rounds, p obtains its critical section. ■

For example, if we choose to build $\mathcal{TC}$ from the leader election algorithm given in [ACD+16] and the token circulation algorithm for arbitrary rooted networks introduced by Cournier et al. in [CDV09], then T_s and T_{tok} are in $O(n)$ rounds, while N_{tok} is in $O(n)$ executions of $PassToken$. Applying these results to Theorem 6.6 shows that the waiting time is achievable in $O(C \times n)$ rounds. Notice also that this implementation of $\mathcal{TC}$ has a memory requirement of $\Theta(\log n)$ bits per process. Hence, $\mathcal{LRA} \circ \mathcal{TC}$ can be implemented using $\Theta(\log n + \log(\max\{|\mathcal{R}_p| : p \in V\}))$ per process p.

6.6.3 Strong Concurrency of $\mathcal{LRA} \circ \mathcal{TC}$

We first prove *No Deadlock*.

Lemma 6.12

Algorithm $\mathcal{LRA} \circ \mathcal{TC}$ meets the No Deadlock *property of strong concurrency: for every subset of processes $X \subseteq V$, for every configuration γ, if $\mathcal{P}_{strong}(X, \gamma)$ holds and $P_{Free}(\gamma) \not\subseteq X$, there exists a configuration γ' and a step $\gamma \mapsto \gamma'$ such that $ContinuousCS(\gamma \dots \gamma') \wedge NoRequest(\gamma \dots \gamma')$.*

Proof: (By the contrapositive.) Assume a configuration γ where no action of the algorithm is enabled. First, if $P_{Free}(\gamma) = \emptyset$, we are done. So from now on, assume $P_{Free}(\gamma) \neq \emptyset$. In γ, $\mathcal{TC}$ has stabilized, by Lemma 6.5. So,

Claim 1: There is a unique tokenholder, t, in γ.

Moreover, as **RsT**-action is disabled at every process, Claim 1 implies:

Claim 2: For every process p, $\gamma(p).token$ if and only if $p = t$.

Claim 3: $\gamma(t).status \neq$ Wait.

Proof of the claim: If $\gamma(t).status =$ Wait, then since $t = Winner(t)$ in γ (by Claim 2), either $\neg IsBlocked(t)$ holds in γ and **E**-action is enabled at t, or **B**-action is enabled at t, a contradiction. □

Claim 4: $\forall p \in P_{Free}(\gamma)$, $\gamma(p).status =$ Blocked.

Proof of the claim: First, by definition, $\gamma(p).status \in \{$Wait, Blocked, Out$\}$ and $\gamma(p).req \neq \perp$. If $\gamma(p).status =$ Out, **R**-action is enabled at p, a contradiction. If $\gamma(p).status =$ Wait, then, $\neg IsBlocked(p)$ since otherwise **B**-action is enabled at p in γ. Consequently, $p \neq Winner(p)$ holds in γ, otherwise **E**-action is enabled at p. So, we can build a sequence of processes $r_0, r_1, \dots, r_k$ where $r_0 = p$ and such that $\forall i \in \{1, \dots, k\}$, $r_i = Winner(r_{i-1})$. (Notice that none of the r_i are the tokenholder, since the tokenholder does not have status Wait, by Claims 1 and 3.) This sequence is finite because $r_0 < r_1 < \dots < r_k$ (so a process cannot be involved several times in this sequence) and the number of processes is finite. Hence, we can take this sequence maximal, in which case, $r_k = Winner(r_k)$ and r_k is then enabled to execute **E**-action, a contradiction. Hence, $\gamma(p).status =$ Blocked. □

Claim 5: $\forall p \in P_{Free}(\gamma)$, $p \in \gamma(t).\mathcal{CN}$ and $\gamma(t).status =$ Blocked.

Proof of the claim: By Claim 4 and since **U**-action is disabled at every process, we have $IsBlocked(p)$ in γ for every process $p \in P_{Free}(\gamma)$. Then, $p \in P_{Free}(\gamma)$ implies $RsrcFree(p)$ in γ, so by Claims 1 and 2, we can conclude. □

Claim 6: There exists a neighbor q of t whose status is In in γ.

Proof of the claim: By Claim 5, $\gamma(t).status =$ Blocked, so $IsBlocked(t)$ holds in γ since **U**-action is disabled at t. Now, by Claim 2, $IsBlocked(t)$ implies that $\neg RsrcFree(t)$ holds in γ, which proves the claim. □

By definition, a consequence of Claim 6 is that q and every process $p \in \gamma(q).\mathcal{CN}$ do not belong to $P_{Free}(\gamma)$. Hence, Claims 5 and 6 imply that $\forall p \in P_{Free}(\gamma)$, $p \in$

$\gamma(t).\mathcal{CN} \setminus (\{q\} \cup \gamma(q).\mathcal{CN})$. So, by letting $X = \gamma(t).\mathcal{CN} \setminus (\{q\} \cup \gamma(q).\mathcal{CN})$, we have $\mathcal{P}_{strong}(X, \gamma)$ and $P_{Free}(\gamma) \subseteq X$, and we are done. ■

We now prove **No Livelock**.

Lemma 6.13

Let $e = (\gamma_j)_{j \geq 0}$ be an execution of $\mathcal{LRA} \circ \mathcal{TC}$, $i \geq 0$ such that $\mathcal{TC}$ is stabilized in γ_i (i is defined by Lemma 6.5), and $t \in V$ the unique tokenholder in γ_i. If $R(e,i,6)$ exists, $R(e,i,6) \leq reqUp(e,i)$, and $\forall j \in \{i+1, \ldots, R(e,i,6)\}$, $PassToken(t)$ is not executed in step $\gamma_{j-1} \mapsto \gamma_j$, then for every $k \in \{R(e,i,4), \ldots, reqUp(e,i)\}$:

- *$\gamma_k(t).req \neq \perp$, $\gamma_k(t).status =$ Blocked, and*
- *$\exists q \in \gamma_k(t).\mathcal{CN}$ such that $\gamma_k(q).status =$ In and $\forall p \in P_{Free}(\gamma_k) \cap \gamma_k(t).\mathcal{CN}$, $p \notin \{q\} \cup \gamma_k(q).\mathcal{CN}$.*

Proof: Let $e = (\gamma_j)_{j \geq 0}$ be an execution of $\mathcal{LRA} \circ \mathcal{TC}$ and let $i \geq 0$ such that $\mathcal{TC}$ has stabilized in γ_i. Let $t \in V$ be the unique tokenholder in γ_i. Assume that $R(e,i,6)$ exists, $R(e,i,6) \leq reqUp(e,i)$, and $\forall j \in \{i+1, \ldots, R(e,i,6)\}$, $PassToken(t)$ is not executed in step $\gamma_{j-1} \mapsto \gamma_j$.

Claim 1: $\forall j \in \{R(e,i,2), \ldots, R(e,i,6)\}$, $\forall p \in V$, $\gamma_j(p).token =$ True if and only if $p = t$.

Proof of the claim: By hypothesis, the value of $TokenReady(p)$ is constant between γ_i and $\gamma_{R(e,i,6)}$. So, if $p.token = TokenReady(p)$ in some configuration γ between γ_i and $\gamma_{R(e,i,2)}$, then $p.token = TokenReady(p)$ holds in all configurations between γ and $\gamma_{R(e,i,6)}$, since **RsT**-action is disabled at p in all these configurations. Since by hypothesis, $TokenReady(p) \equiv (p = t)$ in all configurations between γ_i and $\gamma_{R(e,i,6)}$, we are done.

Assume, otherwise, that $p.token \neq TokenReady(p)$ in all configurations between γ_i and $\gamma_{R(e,i,2)}$, then **RsT**-action (the highest priority action) is continuously enabled at p until p executes it. Now, in this case, p executes it within at most 2 rounds (Remark 6.1), hence, there is a configuration between γ_i and $\gamma_{R(e,i,2)}$, where $p.token = TokenReady(p)$, a contradiction. □

Claim 2: $\gamma_{R(e,i,4)}(t).status =$ Blocked.

Proof of the claim: Assume, by the contradiction, that $\gamma_{R(e,i,4)}(t).status \neq$ Blocked.

a). Assume $\gamma_{R(e,i,2)}(t).status =$ In. If $t.req = \perp$, then $t.req = \perp$ holds in all configurations between γ_i and $\gamma_{R(e,i,6)}$, by hypothesis. Moreover, **RsT**-action is disabled at t in all configurations between $\gamma_{R(e,i,2)}$ and $\gamma_{R(e,i,6)}$, by Claim 1. Hence, by Remark 6.1, t executes **Ex**-action within two rounds from $\gamma_{R(e,i,2)}$, and then **RlT**-action within at most two more rounds. By this latter action, t releases the token by $PassToken(t)$, a contradiction.
Assume now that $t.req \neq \perp$ in $\gamma_{R(e,i,2)}$. Then, by hypothesis, $t.req \neq \perp$ holds in all configurations between $\gamma_{R(e,i,2)}$ and $\gamma_{R(e,i,6)}$. Similarly to the previous case, t releases the token by executing **RlT**-action within two rounds from $\gamma_{R(e,i,2)}$, a contradiction.

b). Assume $\gamma_{R(e,i,2)}(t).status =$ Out. If $t.req = \perp$, then $t.req = \perp$ holds in all configurations between γ_i and $\gamma_{R(e,i,6)}$, by hypothesis. Similarly to

the previous case, t releases the token by executing **RlT**-action within two rounds from $\gamma_{R(e,i,2)}$, a contradiction.
Assume now that $t.req \neq \perp$ in $\gamma_{R(e,i,2)}$. Then, by hypothesis, $t.req \neq \perp$ holds in all configurations between $\gamma_{R(e,i,2)}$ and $\gamma_{R(e,i,6)}$. Moreover, **RsT**-action is disabled at t in all configurations between $\gamma_{R(e,i,2)}$ and $\gamma_{R(e,i,6)}$, by Claim 1. Hence, t sets $t.status$ to WAIT by **R**-action within two rounds from $\gamma_{R(e,i,2)}$ (Remark 6.1). Then, by Claim 1, $t = Winner(t)$ in all subsequent configurations until $\gamma_{R(e,i,6)}$. After executing **R**-action, if $IsBlocked(t)$, then by Claim 1, there exists $q \in \gamma(p).\mathcal{CN}$ such that $q.status =$ IN and $q.req \neq t.req$. By hypothesis, $q.status =$ IN and $q.req \neq t.req$ until at least $\gamma_{reqUp(e,i)}$. So, within at most two more rounds (Remark 6.1), $t.status$ is set to BLOCKED and $t.status$ does not change until at least $\gamma_{reqUp(e,i)}$ (with $R(e,i,6) \leq reqUp(e,i)$) due to q, hence $\gamma_{R(e,i,4)}(t).status =$ BLOCKED, a contradiction. Assume otherwise that $IsBlocked(t)$ does not hold after t executes **R**-action. **E**-action is continuously enabled at t until t executes it, since by Claim 1 $t = Winner(t)$ in all configurations until $\gamma_{R(e,i,6)}$. So, t executes **E**-action within two rounds (Remark 6.1), and by this latter action, t releases the token by $PassToken(t)$, a contradiction.

c). Assume $\gamma_{R(e,i,2)}(t).status =$ WAIT. We obtain a contradiction similarly to the second part of case (b).

d). Assume $\gamma_{R(e,i,2)}(t).status =$ BLOCKED. We obtain a contradiction similarly to the second part of Case (b).

□

Claim 3: $IsBlocked(t)$ in all configurations between $\gamma_{R(e,i,4)}$ and $\gamma_{reqUp(e,i)}$.

Proof of the claim: Assume first there a configuration γ_b between $\gamma_{R(e,i,2)}(t)$ and $\gamma_{R(e,i,4)}(t)$ such that $IsBlocked(t)$ in γ_b. Then, $IsBlocked(t)$ implies $\neg RsrcFree(t)$ in γ_b, by Claim 1. Now, by hypothesis, no process ends its critical section until at least $\gamma_{reqUp(e,i)}$. So, $IsBlocked(t)$ holds in all configurations between γ_b and $\gamma_{reqUp(e,i)}$, and we are done.

Assume otherwise that $\neg IsBlocked(t)$ in every configuration between $\gamma_{R(e,i,2)}(t)$ and $\gamma_{R(e,i,4)}(t)$. t cannot executes **B**-action during $\gamma_{R(e,i,2)}(t) \ldots \gamma_{R(e,i,4)}(t)$. So, if $\gamma_{R(e,i,2)}(t).status \neq$ BLOCKED, then $\gamma_{R(e,i,4)}(t).status \neq$ BLOCKED, contradicting Claim 2. Otherwise, $Unblock(t)$ holds in every configuration between $\gamma_{R(e,i,2)}(t)$ and $\gamma_{R(e,i,4)}(t)$ until $t.status =$ WAIT. Then, as **RsT**-action is disabled at t in all configurations between $\gamma_{R(e,i,2)}$ and $\gamma_{R(e,i,6)}$ (Claim 1), t switches $t.status$ to WAIT by **U**-action before $\gamma_{R(e,i,4)}$ (Remark 6.1) and again $t.status$ remains equal to WAIT until at least $\gamma_{R(e,i,4)}$, contradicting Claim 2. □

By Claims 2 and 3, $t.status =$ BLOCKED in all configurations between $\gamma_{R(e,i,4)}$ and $\gamma_{reqUp(e,i)}$. By Claims 1 and 3, and the hypotheses of the lemma, there is a neighbor q of p such that $q.req \neq t.req$ and $q.status =$ IN in all configurations γ_k between $\gamma_{R(e,i,4)}$ and $\gamma_{reqUp(e,i)}$. By definition, $q \in \gamma_k(t).\mathcal{CN}$, $\gamma_k(t).req \neq \perp$, and $\gamma_k(q).req \neq \perp$. Finally, by definition of P_{Free}, $\forall p \in P_{Free}(\gamma_k) \cap \gamma_k(t).\mathcal{CN}$, $p \notin \{q\} \cup \gamma_k(q).\mathcal{CN}$, indeed no process in P_{Free} can be neighbor of a requesting process with status IN (hence in critical section) using a conflicting resource. ■

Lemma 6.14

Let $e = (\gamma_j)_{j\geq 0}$ be an execution of $\mathcal{LRA} \circ \mathcal{TC}$, $i \geq 0$ such that $\mathcal{TC}$ is stabilized in γ_i (i is defined by Lemma 6.5), and $t \in V$ the unique tokenholder in γ_i. If $R(e,i,4n+2)$ exists and $R(e,i,4n+2) \leq reqUp(e,i)$ and $\forall j \in \{i+1,\ldots,R(e,i,4n+2)\}$, $PassToken(t)$ is not executed in step $\gamma_{j-1} \mapsto \gamma_j$, then,

$$\forall k \in \{R(e,i,4n+2),\ldots,reqUp(e,i)\},\ P_{Free}(\gamma_k) \setminus \gamma_k(t).\mathcal{CN} = \emptyset$$

Proof: Let $e = (\gamma_j)_{j\geq 0}$ be an execution of $\mathcal{LRA} \circ \mathcal{TC}$. Let $i \geq 0$ such that $\mathcal{TC}$ is stabilized in γ_i. Let $t \in V$ the unique tokenholder in γ_i. Assume that $R(e,i,4n+2)$ exists, $R(e,i,4n+2) \leq reqUp(e,i)$, and $\forall j \in \{i+1,\ldots,R(e,i,4n+2)\}$, $PassToken(t)$ is not executed in step $\gamma_{j-1} \mapsto \gamma_j$.

Claim 1: $\forall j \in \{R(e,i,2),\ldots,R(e,i,4n+2)\}$, $\forall p \in V$, $\gamma_j(t).token = \textsc{True}$ if and only if $p = t$, and **RsT**-action is disabled at p in γ_j.

Proof of the claim: Identical to the proof of Claim 1 in Lemma 6.13. □

Then, P_{Free} contains only requesting processes p ($p.req \neq \perp$) with no neighbor q using a resource conflicting with the requested one (namely, such that $q.status = \textsc{In}$ and $q.req \not\approx p.req$). So, no process can enter P_{Free} during $\gamma_i \ldots reqUp(e,i)$ since no new request occurs and no critical section is released.

Let $j \in \{R(e,i,2),\ldots,R(e,i,4(n-1)+2)\}$. If $P_{Free}(\gamma_j) \setminus \gamma_j(t).\mathcal{CN}$ is empty, then it remains so until $\gamma_{R(e,i,4n+2)}$. Otherwise, let $q = \max\{x \in P_{Free}(\gamma_j) \setminus \gamma_j(p).\mathcal{CN}\}$. In the worst case, q has status Out, it reaches status Wait in at most 2 rounds (by Claim 1 and Remark 6.1). Either q exited P_{Free} in the meantime, *i.e.*, a process with status Wait entered its critical section meanwhile and is using a conflicting resource, or q reaches status In (using **E**-action) in at most 2 additional rounds (by Claim 1 and Remark 6.1). Indeed, in the latter case, $IsBlocked(q)$ does not hold since $q \in P_{Free}$ ensures that $RsrcFree(q)$ and since it has no conflicting neighbor holding the token by assumption; furthermore, $q = Winner(q)$ by definition. Hence, at most 4 rounds later, q has exited P_{Free}.

Repeating the reasoning n times ensures that in configuration $\gamma_{R(e,i,4n+2)}$ the set $P_{Free}(\gamma_{R(e,i,4n+2)}) \setminus \mathcal{CN}_t(\gamma_{R(e,i,4n+2)})$ is empty. Then, as long as no critical section is released and no new request occurs, P_{Free} remains empty. ■

Lemma 6.15

Let $e = (\gamma_j)_{j\geq 0}$ be an execution of $\mathcal{LRA} \circ \mathcal{TC}$ and $i \geq 0$ such that $\mathcal{TC}$ is stabilized in γ_i (i is defined by Lemma 6.5). If $R(e,i,6n(N_{tok}+1))$ exists and $R(e,i,6n(N_{tok}+1)) \leq reqUp(e,i)$, then

- *either for every $k \in \{R(e,i,6n(N_{tok}+1)),\ldots,reqUp(e,i)\}$, $P_{Free}(\gamma_k) = \emptyset$, or*
- *for every $k \in \{R(e,i,6n(N_{tok}+1)-6),\ldots,reqUp(e,i)-1\}$, $PassToken$ is not executed in step $\gamma_k \mapsto \gamma_{k+1}$.*

Proof: Let $e = (\gamma_j)_{j\geq 0}$ be an execution of $\mathcal{LRA} \circ \mathcal{TC}$. Let $i \geq 0$ such that $\mathcal{TC}$has stabilized at γ_i. Assume that $R(e,i,6n(N_{tok}+1))$ exists and $R(e,i,6n(N_{tok}+1)) \leq reqUp(e,i)$.

Similarly to the proof of Lemma 6.14, P_{Free} cannot increase, hence if it is empty at some configuration γ_k with $k \in \{R(e,i,6n(N_{tok}+1)), \dots, reqUp(e,i)\}$, we are done.

Let $k \in \{R(e,i,6n(N_{tok}+1)), \dots, reqUp(e,i)\}$. Assume $P_{Free}(\gamma_k) \neq \emptyset$ and let $p \in P_{Free}(\gamma_k)$. We deduce from Lemma 6.13, that if $PassToken$ has not been executed by the tokenholder, during 6 consecutive rounds, then the token will stay at this process until $reqUp(e,i)$. Furthermore, properties of $\mathcal{TC}$ ensures that after at most N_{tok} executions of $PassToken$ the token will reach p. Then, at the latest, at configuration $\gamma_{R(e,k,6N_{tok})}$ ($6N_{tok}$ rounds later), the token is either blocked until $reqUp(e,i)$ at some process (but not p) or has passed through p. Let consider the second case: if when the token is at p, P_{Free} still contains p, then, after at most 6 additional rounds (still by Lemma 6.13), p has access to critical section and exits P_{Free}.

Repeating this reasoning n times, we have that in at most $6n(N_{tok}+1)$ rounds, either P_{Free} is empty or the token is blocked until $reqUp(e,i)$. ■

Lemma 6.16

Algorithm $\mathcal{LRA} \circ \mathcal{TC}$ meets the No Livelock *property of strong concurrency: there exists a number of rounds $T_{PC} > 0$ such that for every execution $e = (\gamma_i)_{i\geq 0}$ and for every index $i \geq 0$, if $R(e,i,T_{PC})$ exists, then*

$$R(e,i,T_{PC}) \leq reqUp(e,i) \Rightarrow \exists X, \mathcal{P}_{strong}(X, \gamma_{R(e,i,T_{PC})}) \wedge P_{Free}(\gamma_{R(e,i,T_{PC})}) \subseteq X$$

Proof: We pose $T_{PC} = T_{tok} + 6n(N_{tok}+1) + 4n - 4$. Let $e = (\gamma_i)_{i\geq 0}$ be an execution of $\mathcal{LRA} \circ \mathcal{TC}$ and let $i \geq 0$. Assume that $R(e,i,T_{PC})$ exists and $R(e,i,T_{PC}) \leq reqUp(e,i)$. After T_{tok} rounds, $\mathcal{TC}$ has stabilized. Using Lemma 6.15, we have two cases:

1. After $T_{tok} + 6n(N_{tok}+1)$, P_{Free} is empty and remains so until $reqUp(e,i)$. In this case, we are done.
2. For every $k \in \{R(e,i,6n(N_{tok}+1)-6), \dots, reqUp(e,i)-1\}$, $PassToken$ is not executed in step $\gamma_k \mapsto \gamma_{k+1}$. Note that this implies that $PassToken$ is not executed during the last 6 rounds by the tokenholder t. This allows to apply Lemma 6.13: there exists a conflicting neighbor of t, q, such that $\forall p \in P_{Free} \cap \gamma_k(t).\mathcal{CN}$, $p \notin \{q\} \cup \gamma_k(q).\mathcal{CN}$.

   As t holds the token from configuration $R(e,i,6n(N_{tok}+1)-6)$ to configuration $reqUp(e,i)$, and as $R(e,i,T_{tok}+6n(N_{tok}+1)+4n-4) \leq reqUp(e,i)$, we can apply Lemma 6.14 between configuration $R(e,i,T_{tok}+6n(N_{tok}+1)-6)$ and $R(e,i,T_{tok}+6n(N_{tok}+1)+4n-4)$: this proves that

$$P_{Free}(\gamma_{R(e,i,T_{tok}+6n(N_{tok}+1)+4n-4)}) \setminus \gamma_{R(e,i,T_{tok}+6n(N_{tok}+1)+4n-4)}(t).\mathcal{CN} = \emptyset$$

■

By Lemmas 6.12 and 6.16, follows.

Theorem 6.7

Algorithm $\mathcal{LRA} \circ \mathcal{TC}$ is strongly concurrent.

6.7 Conclusion

Summary of Contributions. In this chapter, we have characterized the maximal level of concurrency we can obtain in resource allocation problems by proposing the notion of *maximal concurrency*. This notion is versatile, *e.g.*, it generalizes the avoiding ℓ-deadlock [FLBB79] and (strict) (k,ℓ)-liveness [DHV03] defined for the ℓ-exclusion and k-out-of-ℓ-exclusion problems, respectively.

From [FLBB79], we already know that *maximal concurrency* can be achieved in some important global resource allocation problems.[3] Now, perhaps surprisingly, our results show that *maximal concurrency* cannot be achieved in problems that can be expressed with the Local Ressource Allocation paradigm. However, we have shown that *strong concurrency* (a high, but not maximal, level of concurrency) can be achieved by a snap-stabilizing LRA algorithm, called $\mathcal{LRA} \circ \mathcal{TC}$. We have to underline that the level of concurrency we achieve here is similar to the one obtained in the committee coordination problem [BDP11].

Perspectives. As a future work, defining the exact class of resource allocation problems where *maximal concurrency* (resp. *strong concurrency*) can be achieved is a challenging perspective.

The drawback of our highly concurrent algorithm is its waiting time ($\Theta(n)$ rounds). Now, designing a highly concurrent algorithm with a tighter waiting time seems to be difficult, even maybe impossible. Indeed, it has been shown that the maximal concurrency and a fast waiting time are incompatible in the ℓ-exclusion problem [CDDL15]. Precisely, Carrier et al. have shown in [CDDL15] that it is possible to design a ℓ-exclusion algorithm that is either maximal concurrent, or asymptotically optimal in waiting time ($O(\lceil \frac{n}{\ell} \rceil)$ rounds), but obtaining an algorithm which achieves both properties is impossible. We might expect that this latter result can be extended to show the incompatibility of strong concurrency and fast waiting time in other resource allocation problems, such as LRA.

[3] By "global" we mean resource allocation problems where a resource can be accessed by any process.

Chapter 7

Conclusion

"Go now. Our journey is done. And may we meet again, in the clearing, at the end of the path"

— Stephen King, *The Dark Tower*

Contents

7.1 Thesis Contributions

In this thesis, we have studied classical problems of distributed computing under uncertain contexts. By uncertain context, we mean that the context of execution of the distributed system is not completely known *a priori* or is unsettled. We focused on two kinds of uncertainty: incomplete identification of the processes and presence of faults. More precisely, we explored two axes of research:

- going towards more anonymity by proposing deterministic algorithms for networks of homonyms and anonymous processes,
- ensuring more safety guarantees during the convergence of deterministic self-stabilizing algorithms

with efficient solutions.

7.1.1 Chapter 3 – Leader Election in Unidirectional Rings with Homonym Processes

The first contributions of this thesis focus on leader election in static unidirectional rings with homonym processes, *i.e.*, processes are identified but several processes may have the same ID, called label. We considered here the message-passing model.

We have proven that message-terminating leader election is impossible to solve in a unidirectional ring with a symmetric labeling. Thus, we have considered unidirectional rings with an asymmetric labeling. We have showed that it is impossible to solve process-terminating leader election in the class $\mathcal{U}^*$ that contains every unidirectional rings with at least one unique label. As a consequence, it is impossible to solve process-terminating leader election in the class $\mathcal{A}$ of all unidirectional rings with an asymmetric labeling. We have also proven that it is impossible to solve message-terminating leader election in the class $\mathcal{K}_k$ that contains unidirectional rings with no more than $k \geq 1$ processes with the same label since $\mathcal{K}_k$ contains symmetric rings. More precisely, k is an upper bound on the multiplicity of the labels, *i.e.*, the number of processes that have the same label (k is known by the processes).

Then, we have proposed three algorithms U_k, A_k, and B_k. Algorithm U_k is a process-terminating leader election algorithm for class $\mathcal{U}^* \cap \mathcal{K}_k$, for any $k \geq 1$. It is asymptotically optimal in time with $\Theta(kn)$ time units and in memory since it requires $O(\log k+b)$ bits per processes, where b is the number of bits required to store a label. Its message complexity is $O(kn)$. Algorithms A_k and B_k both solves the process-terminating leader election problem for class $\mathcal{A} \cap \mathcal{K}_k$, for any $k \geq 1$. Algorithm A_k is asymptotically optimal in time ($\Theta(kn)$ time units) but requires $O(knb)$ bits per process and $O(kn^2)$ sending of messages. On the contrary, Algorithm B_k is asymptotically optimal in memory ($O(\log k + b)$ bits per process) but its time complexity is $O(k^2n^2)$ and its message complexity is $O(k^2n^2)$.

7.1.2 Chapter 4 – Self-stabilizing Leader Election under Unfair Daemon

Similarly to Chapter 3, we have considered the leader election problem in Chapter 4, yet in a different context. We proposed a silent self-stabilizing leader election algorithm, called $\mathcal{LE}$, for any static and identified network of arbitrary connected and bidirectional topology. Algorithm $\mathcal{LE}$ is written in the locally shared memory model, requires no global knowledge on the network (*e.g.*, no upper bound on the number of processes or the diameter), and assumes the distributed unfair daemon.

From an arbitrary configuration, Algorithm $\mathcal{LE}$ converges to a terminal configuration in at most $3n+\mathcal{D}$ rounds, where $\mathcal{D}$ is the diameter of the network, and we built a network for any $n \geq 4$ and any $\mathcal{D}$ such that $2 \leq \mathcal{D} \leq n-2$ in which there is a possible execution that lasts exactly $3n+\mathcal{D}$ rounds. In this terminal configuration, every process knows the ID of the leader, and a spanning tree rooted at the leader is defined. Algorithm $\mathcal{LE}$ is asymptotically optimal in memory with $\Theta(\log n)$ bits per processes.

We have showed that Algorithm $\mathcal{LE}$ stabilizes in a polynomial number of steps. Indeed, it converges in $\Theta(n^3)$ steps. For fair comparison, we studied the step complexity of the previous best algorithms with similar settings, *i.e.*, no global knowledge required, proven under a distributed unfair daemon. For any $n \geq 5$, we have showed that there exists a network in which there exists an execution of the algorithm proposed in [DLV11a], denoted here $\mathcal{DLV}_1$, that stabilizes in $\Omega(2^{\lfloor \frac{n-1}{4} \rfloor})$ steps. Similarly, we proved that for a given $\alpha \geq 3$, for any $\beta \geq 2$, there exists a network of $n = 2^\alpha \times \beta$ processes, in which a possible execution of the algorithm proposed in [DLV11b], denoted here $\mathcal{DLV}_2$, stabilizes in

$\Omega(n^{\alpha+1})$. Hence, the stabilization times of $\mathcal{DLV}_1$ and $\mathcal{DLV}_2$ in steps are not polynomial.

7.1.3 Chapter 5 – Gradual Stabilization under (τ, ρ)-dynamics and Unison

In Chapter 5, we proposed a variant of self-stabilization, called gradual stabilization under (τ, ρ)-dynamics. This variant is especially designed for dynamic networks. Indeed, an algorithm is gradually stabilizing under (τ, ρ)-dynamics if it is self-stabilizing and satisfies the following additional feature. After up to τ dynamic steps of type ρ starting from a legitimate configuration, a gradually stabilizing algorithm first quickly recovers a configuration from which a minimum safety guarantee is satisfied. Then, it gradually converges to specifications offering stronger and stronger quality of service, until recovering a configuration from which the two following conditions hold. Its initial specification is satisfied and, if up to τ ρ-dynamic steps hit the system again, it is ready to achieve gradual convergence.

We have illustrated this new property by proposing a gradually stabilizing algorithm denoted $\mathcal{DSU}$ for the unison problem. $\mathcal{DSU}$ is designed in the locally shared memory model for any arbitrary anonymous network that is initially connected and assumes the distributed unfair daemon. It is gradually stabilizing under $(1, \mathsf{BULCC})$-dynamics. A BULCC-dynamic step contains a finite yet unbounded number of topological changes such that, after such a step, the network: (1) contains at most N processes (where N is an upperbound on the number of processes in the system at any time), (2) is connected, and (3) if the clock period α is greater than 3, every process that joined the system should be linked to at least one process that was already in the system before the dynamic step, except if all those processes have left the system. (We have studied the necessity of these conditions.) Starting from a configuration satisfying the strong unison (there is at most two different yet consecutive clock values), if a BULCC-dynamic step hits the system, $\mathcal{DSU}$ immediately satisfies the partial unison (the clocks of two neighboring processes differ of at most one increment, except for incoming processes). Then, in one round, it satisfies the weak unison (the clocks of every two neighboring processes differ of at most one increment) and converges to strong unison in $(\mu + 1)\mathcal{D}_1 + 2$ rounds, where μ is a parameter greater or equal than $\max(2, N)$, and $\mathcal{D}_1$ is the diameter of the network after the dynamic step.

7.1.4 Chapter 6 – Concurrency in Local Resource Allocation

Finally, in Chapter 6, we have proposed a property called maximal concurrency to characterize the maximal level of concurrency that can be achieved in resource allocation problems. This notion generalizes similar notions previously defined for specific problems, *e.g.*, the avoiding ℓ-deadlock [FLBB79] defined for the ℓ-exclusion problem and the (strict) (k, ℓ)-liveness [DHV03] defined for the k-out-of-ℓ exclusion problem.

We showed that, even if maximal-concurrency can be achieved in some problems such as the ℓ-exclusion [FLBB79], it cannot be achieved in a wide class of resource allocation problems called Local Resource Allocation (LRA). Nonetheless, we proved that the strong concurrency, a high but not maximal level of concurrency can be achieved in the LRA

problem. More precisely, we have proposed a strongly concurrent snap-stabilizing LRA algorithm, called $\mathcal{LRA}$, for connected bidirectional networks of arbitrary topology. $\mathcal{LRA}$ is written in the locally shared memory model and assumes a weakly fair daemon.

7.2 General Perspectives

Detailed perspectives are already presented in the conclusion of each contribution chapter. Thus, in this section, we propose more general perspectives. There are three main axes of future research.

Homonyms and Self-stabilization. The model of homonym processes has only been slightly studied. In particular, in our knowledge, no self-stabilizing algorithm has been proposed for networks containing homonym processes. However, our results of Chapter 3, in particular Algorithm A_k, seems very promising for an adaptation to self-stabilization. It is further research to design self-stabilizing algorithms for homonym networks, first for leader election, and then for other problems.

Gradual Stabilization. In Chapter 5, we propose a new property, the gradual stabilization under (τ, ρ)-dynamics, and we illustrate this property for $\tau = 1$ with a unison algorithm. The generalization for $\tau > 1$ remains open. Moreover, achieving this property for other (dynamic) problems is a natural extension that could lead to a deeper understanding and then allows the generalization of the approach by the design of a transformer (*i.e.*, an algorithm that supplies the gradual stabilization property to merely self-stabilizing algorithms).

Concurrency. As stated in Chapter 6, concurrency is an issue in resource allocation problems that has not been extensively studied. However, it is fundamental to maximize the usage of resources and minimize the waiting time of requesting processes. Maybe surprisingly, it has been proven that the maximal level of concurrency (called here maximal-concurrency) cannot be achieved in various problems, *i.e.*, k-out-of-ℓ exclusion, committee coordination, and local resource allocation (see Chapter 6). The only problems for which we know that maximal-concurrency can be achieved are ℓ-exclusion problem [FLBB79] and trivially the mutual exclusion problem. Thus, the level of concurrency that can be achieved in other resource allocation problems, for instance, group mutual exclusion or drinking philosophers problem, is worth investigating.

Bibliography

[AB93] Yehuda Afek and Geoffrey M. Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7(1):27–34, 1993. (Cited p. 20.)

[ABB98] Yehuda Afek and Anat Bremler-Barr. Self-stabilizing unidirectional network algorithms by power supply. *Chicago Journal of Theoretical Computer Science*, 1998, 1998. (Cited on pp. 21 and 74.)

[ACD+14] Karine Altisen, Alain Cournier, Stéphane Devismes, **Anaïs Durand**, and Franck Petit. Self-stabilizing leader election in polynomial steps. In *Proceedings of the 16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'14)*, pages 106–119, Paderborn, Germany, September 28 - October 1, 2014. (Cited on pp. 21 and 75.)

[ACD+15] Karine Altisen, Alain Cournier, Stéphane Devismes, **Anaïs Durand**, and Franck Petit. Élection autostabilisante en un nombre polynomial de pas de calcul. In *Proceedings des 17èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications (ALGOTEL'15)*, Beaune, France, June 2-5, 2015. (Cited on pp. 21 and 75.)

[ACD+16] Karine Altisen, Alain Cournier, Stéphane Devismes, **Anaïs Durand**, and Franck Petit. Self-stabilizing leader election in polynomial steps. *Information and Computation*, 2016. (Cited on pp. 20, 21, 75, 187, 198, and 226.)

[AD14] Karine Altisen and Stéphane Devismes. On probabilistic snap-stabilization. In *Proceedings of the 15th International Conference on Distributed Computing and Networking (ICDCN'14)*, pages 272–286, Coimbatore, India, January 4-7, 2014. (Cited p. 173.)

[ADD15] Karine Altisen, Stéphane Devismes, and **Anaïs Durand**. Concurrency in snap-stabilizing local resource allocation. In *Proceedings of the 3rd International Conference on Networked Systems (NETYS'15)*, pages 77–93, Agadir, Morocco, May 13-15, 2015. (Cited on pp. 22 and 172.)

[ADD+16a] Karine Altisen, Ajoy K. Datta, Stéphane Devismes, **Anaïs Durand**, and Lawrence L. Larmore. Leader election in rings with bounded multiplicity

(short paper). In *Proceedings of the 18th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'16)*, pages 1–6, Lyon, France, November 7-10, 2016. (Cited on pp. 21 and 42.)

[ADD16b] Karine Altisen, Stéphane Devismes, and **Anaïs Durand**. Concurrence et allocation de ressources locales instantanément stabilisante. In *Proceedings of the 18èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications (ALGOTEL'16)*, Bayonne, France, May 24-27, 2016. (Cited on pp. 22 and 172.)

[ADD+17a] Karine Altisen, Ajoy K. Datta, Stéphane Devismes, **Anaïs Durand**, and Lawrence L. Larmore. Leader election in asymmetric labeled unidirectional rings. In *Proceedings of the 31st International Parallel and Distributed Processing Symposium (IPDPS'17)*, pages 182–191, Orlando, Florida, USA, May 29 - June 2, 2017. (Cited on pp. 21 and 42.)

[ADD17b] Karine Altisen, Stéphane Devismes, and **Anaïs Durand**. Concurrency in snap-stabilizing local resource allocation. *Journal of Parallel and Distributed Computing*, 102:42–56, 2017. (Cited on pp. 22 and 172.)

[ADDP16] Karine Altisen, Stéphane Devismes, Anaïs Durand, and Franck Petit. Gradual stabilization under τ-dynamics. In *Proceedings of the 22nd International Conference on Parallel and Distributed Computing (Euro-Par'16)*, pages 588–602, Grenoble, France, August 24-26, 2016. (Cited on pp. 22 and 130.)

[ADG91] Anish Arora, Shlomi Dolev, and Mohamed G. Gouda. Maintaining digital clocks in step. *Parallel Processing Letters*, 1:11–18, 1991. (Cited p. 131.)

[AG94] Anish Arora and Mohamed G. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9):1026–1038, 1994. (Cited on pp. 74 and 187.)

[AKM+93] Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. Time optimal self-stabilizing synchronization. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC'93)*, pages 652–661, San Diego, California, USA, May 16-18 1993. (Cited on pp. 74 and 128.)

[AKM+07] Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. A time-optimal self-stabilizing synchronizer using a phase clock. *IEEE Transactions on Dependable and Secure Computing*, 4(3):180–190, 2007. (Cited p. 21.)

[AN05] Anish Arora and Mikhail Nesterenko. Unifying stabilization and termination in message-passing systems. *Distributed Computing*, 17(3):279–290, 2005. (Cited p. 21.)

[Ang80] Dana Angluin. Local and global properties in networks of processors. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing (STOC'80)*, pages 82–93, Los Angeles, California, USA, April 28-30, 1980. (Cited p. 40.)

[APHV00] Alan D. Amis, Ravi Prakash, Dung Huynh, and Thai Vuong. Max-min d-cluster formation in wireless ad hoc networks. In *Proceedings of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies, Reaching the Promised Land of Communications (INFOCOM'00)*, pages 32–41, Tel Aviv, Israel, March 26-30, 2000. (Cited p. 13.)

[APV91] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction (extended abstract). In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science (FOCS'91)*, pages 268–277, San Juan, Puerto Rico, October 1-4, 1991. (Cited p. 21.)

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, 1985. (Cited p. 14.)

[BCV03] Lélia Blin, Alain Cournier, and Vincent Villain. An improved snap-stabilizing PIF algorithm. In *Proceedings of the 6th International Symposium on Self-Stabilizing Systems (SSS'03)*, pages 199–214, San Francisco, California, USA, June 24-25, 2003. (Cited p. 80.)

[BDKP16] Nicolas Braud-Santoni, Swan Dubois, Mohamed-Hamza Kaaouachi, and Franck Petit. The next 700 impossibility results in time-varying graphs. *International Journal of Networking and Computing*, 6(1):27–41, 2016. (Cited p. 33.)

[BDP11] Borzoo Bonakdarpour, Stéphane Devismes, and Franck Petit. Snap-stabilizing comittee coordination. In *Proceedings of the 25th IEEE International Symposium on Parallel and Distributed Processing (IPDPS'11)*, pages 231–242, Anchorage, Alaska, USA, May 16-20. 2011. (Cited on pp. 171 and 204.)

[BDPV07] Alain Bui, Ajoy K. Datta, Franck Petit, and Vincent Villain. Snap-stabilization and PIF in tree networks. *Distributed Computing*, 20(1):3–19, 2007. (Cited on pp. 19, 36, 37, 172, and 228.)

[BEF84] Dennis J. Baker, Anthony Ephremides, and Julia A. Flynn. The design and simulation of a mobile radio network with distributed control. *IEEE Journal on Selected Areas in Communications*, 2(1):226–237, 1984. (Cited p. 13.)

[Ben83] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing (PODC'83)*, pages 27–30, Montreal, Quebec, Canada, August 17-19, 1983. (Cited p. 17.)

[BGK98] Joffroy Beauquier, Christophe Genolini, and Shay Kutten. k-stabilization of reactive tasks. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (PODC'98)*, page 318, Puerto Vallarta, Mexico, June 28 - July 2, 1998. (Cited p. 19.)

[BGM93] James E. Burns, Mohamed G. Gouda, and Raymond E. Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7(1):35–42, 1993. (Cited p. 19.)

[BK07] Janna Burman and Shay Kutten. Time optimal asynchronous self-stabilizing spanning tree. In *Proceedings of the 21st International Symposium on Distributed Computing (DISC'07)*, pages 92–107, Lemesos, Cyprus, September 24-26, 2007. (Cited p. 74.)

[Bor26] Otakar Borůvka. O jistém problému minimálním. *Práce mor. přírodověd. spol. v Brně III*, 3:37–58, 1926. In Czech and German. (Cited p. 13.)

[Bou07] Christian Boulinier. *L'Unisson*. PhD thesis, Université de Picardie Jules Vernes, France, 2007. Available online: https://tel.archives-ouvertes.fr/tel-01511431. (Cited on pp. 131, 145, 146, 147, 148, and 157.)

[BPR13] Lélia Blin, Maria Potop-Butucaru, and Stephane Rovedakis. A super-stabilizing $\log(n)$-approximation algorithm for dynamic Steiner trees. *Theoretical Computer Science*, 500:90–112, 2013. (Cited p. 131.)

[BPRT10] Lélia Blin, Maria Gradinariu Potop-Butucaru, Stephane Rovedakis, and Sébastien Tixeuil. Loop-free super-stabilizing spanning tree construction. In *Proceedings of the 12th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'10)*, pages 50–64, New York, New York, USA, September 20-22, 2010. (Cited p. 131.)

[BPV04] Christian Boulinier, Franck Petit, and Vincent Villain. When graph theory helps self-stabilization. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC'04)*, pages 150–159, St. John's, Newfoundland, Canada, July 25-28, 2004. (Cited on pp. 131, 170, and 171.)

[BT17a] Lélia Blin and Sébastien Tixeuil. Compact deterministic self-stabilizing leader election on a ring: the exponential advantage of being talkative. *Distributed Computing*, pages 1–28, 2017. (Cited p. 74.)

[BT17b] Lélia Blin and Sébastien Tixeuil. Compact self-stabilizing leader election for arbitrary networks. Online, February 24, 2017. arxiv:1702.07605[cs.DC]. (Cited p. 74.)

[CDD+15] Fabienne Carrier, Ajoy Kumar Datta, Stéphane Devismes, Lawrence L. Larmore, and Yvan Rivierre. Self-stabilizing (f, g)-alliances with safe convergence. *Journal of Parallel and Distributed Computing*, 81-82:11–23, 2015. (Cited p. 130.)

[CDDL15] Fabienne Carrier, Ajoy Kumar Datta, Stéphane Devismes, and Lawrence L. Larmore. Self-stabilizing ℓ-exclusion revisited. In *Proceedings of the 16th International Conference on Distributed Computing and Networking (ICDCN'15)*, pages 3:1–3:10, Goa, India, January 4-7, 2015. (Cited p. 204.)

[CDP03] Sébastien Cantarell, Ajoy Kumar Datta, and Franck Petit. Self-stabilizing atomicity refinement allowing neighborhood concurrency. In *Proceedings of the 6th International Symposium on Self-Stabilizing Systems (SSS'03)*, pages 102–112, San Francisco, California, USA, June 24-25, 2003. (Cited on pp. 170, 171, and 172.)

[CDV09] Alain Cournier, Stéphane Devismes, and Vincent Villain. Light enabling snap-stabilization of fundamental protocols. *ACM Transactions on Autonomous and Adaptive Systems*, 4(1), 2009. (Cited on pp. 187 and 198.)

[CFG92] Jean-Michel Couvreur, Nissim Francez, and Mohamed G. Gouda. Asynchronous unison (extended abstract). In *Proceedings of the 12th International Conference on Distributed Computing Systems (ICDCS'92)*, pages 486–493, Yokohama, Japan, June 9-12, 1992. (Cited on pp. 14, 131, 132, and 146.)

[CFQS12] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012. (Cited p. 33.)

[Cha82] Ernest J. H. Chang. Echo algorithms: Depth parallel operations on general graphs. *IEEE Transactions on Software Engineering*, 8(4):391–401, 1982. (Cited on pp. 11 and 80.)

[Cha93] Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, 1993. (Cited p. 12.)

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996. (Cited p. 17.)

[CR79] Ernest J. H. Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22(5):281–283, 1979. (Cited p. 40.)

[CT91] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems (preliminary version). In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing (PODC'91)*, pages 325–340, Montreal, Quebec, Canada, August 19-21, 1991. (Cited p. 17.)

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996. (Cited p. 17.)

[DDPT11] Shlomi Dolev, Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. Stabilizing data-link over non-fifo channels with optimal fault-resilience. *Information Processing Letters*, 111(18):912–920, 2011. (Cited p. 20.)

[DDT06] Sylvie Delaët, Bertrand Ducourthial, and Sébastien Tixeuil. Self-stabilization with r-operators revisited. *Journal of Aerospace Computing, Information, and Communication*, 3(10):498–514, 2006. (Cited p. 21.)

[DFT14] Carole Delporte-Gallet, Hugues Fauconnier, and Hung Tran-The. Leader election in rings with homonyms. In *Proceedings of the 2nd International Conference on Networked Systems (NETYS'14)*, pages 9–24, Marrakech, Morocco, May 15-17, 2014. (Cited on pp. 40 and 41.)

[DGS99] Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory requirements for silent stabilization. *Acta Informatica*, 36(6):447–462, 1999. (Cited p. 74.)

[DH97] Shlomi Dolev and Ted Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 1997, 1997. (Cited on pp. 19, 74, 128, 130, 131, 134, 187, and 228.)

[DHV03] Ajoy K. Datta, Rachid Hadid, and Vincent Villain. A new self-stabilizing k-out-of-ℓ exclusion algorithm on rings. In *Proceedings of the 6th International Symposium on Self-Stabilizing Systems (SSS'03)*, pages 113–128, San Francisco, California, USA, June 24-25, 2003. (Cited on pp. 171, 178, 179, 180, 204, 207, and 231.)

[Dij65] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965. (Cited on pp. 12 and 170.)

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974. (Cited on pp. 17, 18, 35, 37, 127, and 228.)

[Dij78] Edsger W. Dijkstra. Two starvation-free solutions of a general exclusion problem. Technical Report EWD 625, Plataanstraat 5, 5671, AL Nuenen, The Netherlands, 1978. (Cited on pp. 12 and 170.)

[Dij86] Edsger W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1(1):5–6, 1986. (Cited on pp. 18 and 228.)

[DIM97] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Uniform dynamic self-stabilizing leader election. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):424–440, 1997. (Cited p. 36.)

[DJPV00] Ajoy K. Datta, Colette Johnen, Franck Petit, and Vincent Villain. Self-stabilizing depth-first token circulation in arbitrary rooted networks. *Distributed Computing*, 13(4):207–218, 2000. (Cited p. 187.)

[DLP10] Ajoy Kumar Datta, Lawrence L. Larmore, and Hema Piniganti. Self-stabilizing leader election in dynamic networks. In *Proceedings of the 12th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'10)*, pages 35–49, New York, New York, USA, September 20-22, 2010. (Cited p. 74.)

[DLS88] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988. (Cited p. 17.)

[DLV11a] Ajoy K. Datta, Lawrence L. Larmore, and Priyanka Vemula. An $o(n)$-time self-stabilizing leader election algorithm. *Journal of Parallel and Distributed Computing*, 71(11):1532–1544, 2011. (Cited on pp. 74, 75, 108, 109, 126, 187, 206, 226, and 230.)

[DLV11b] Ajoy K. Datta, Lawrence L. Larmore, and Priyanka Vemula. Self-stabilizing leader election in optimal space under an arbitrary scheduler. *Theoretical Computer Science*, 412(40):5541–5561, 2011. (Cited on pp. 74, 75, 114, 115, 126, 187, 206, 226, and 230.)

[Dol00] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000. (Cited p. 185.)

[DP04] Stefan Dobrev and Andrzej Pelc. Leader election in rings with nonunique labels. *Fundamenta Informaticae*, 59(4):333–347, 2004. (Cited on pp. 40 and 41.)

[DP16] Dariusz Dereniowski and Andrzej Pelc. Topology recognition and leader election in colored networks. *Theoretical Computer Science*, 621:92–102, 2016. (Cited on pp. 40 and 41.)

[DT11] Swan Dubois and Sébastien Tixeuil. A taxonomy of daemons in self-stabilization. Online, October 3, 2011. arxiv:1110.0334[cs.DC]. (Cited p. 32.)

[Dur17] Anaïs Durand. Élection et anneaux unidirectionnels en présence d'homonymes. In *Proceedings of the 19èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications (ALGOTEL'17)*, Quiberon, France, May 29-June 2, 2017. (Cited on pp. 21 and 42.)

[FKK+04] Paola Flocchini, Evangelos Kranakis, Danny Krizanc, Flaminia L. Luccio, and Nicola Santoro. Sorting and election in anonymous asynchronous rings. *Journal of Parallel and Distributed Computing*, 64(2):254–265, 2004. (Cited on pp. 40 and 41.)

[FLBB79] Michael J. Fischer, Nancy A. Lynch, James E. Burns, and Allan Borodin. Resource allocation with immunity to limited process failure (preliminary report). In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science (FOCS'79)*, pages 234–254, San Juan, Puerto Rico, October 29-31, 1979. (Cited on pp. 12, 170, 171, 175, 178, 180, 204, 207, 208, 231, and 232.)

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985. (Cited on pp. 13, 17, and 225.)

[GGHP96] Sukumar Ghosh, Arobinda Gupta, Ted Herman, and Sriram V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*, pages 45–54, Philadelphia, Pennsylvania, USA, May 23-26, 1996. (Cited on pp. 19, 128, and 134.)

[GH90] Mohamed G. Gouda and Ted Herman. Stabilizing unison. *Information Processing Letters*, 35(4):171–175, 1990. (Cited p. 131.)

[GH07] Mohamed G. Gouda and F. Furman Haddix. The alternator. *Distributed Computing*, 20(1):21–28, 2007. (Cited on pp. 170 and 171.)

[GM91] Mohamed G. Gouda and Nicholas J. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, 1991. (Cited p. 20.)

[Gou01] Mohamed G. Gouda. The theory of weak stabilization. In *Proceedings of the 5th International Workshop on Self-Stabilizing Systems (WSS'01)*, pages 114–123, Lisbon, Portugal, October 1-2, 2001. (Cited p. 19.)

[GP93] Ajei S. Gopal and Kenneth J. Perry. Unifying self-stabilization and fault-tolerance (preliminary version). In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC'93)*, pages 195–206, Ithaca, New York, USA, August 15-18, 1993. (Cited on pp. 18 and 228.)

[GT02] Christophe Genolini and Sébastien Tixeuil. A lower bound on dynamic k-stabilization in asynchronous systems. In *Proceedings of the 21st Symposium on Reliable Distributed Systems (SRDS'02)*, page 212, Osaka, Japan, October 13-16, 2002. (Cited p. 128.)

[GT07] Maria Gradinariu and Sébastien Tixeuil. Conflict managers for self-stabilization without fairness assumption. In *Proceedings of the 27th IEEE International Conference on Distributed Computing Systems (ICDCS'07)*, page 46, Toronto, Ontario, Canada, June 25-29, 2007. (Cited p. 171.)

[HC93] Shing-Tsaan Huang and Nian-Shing Chen. Self-stabilizing depth-first token circulation on networks. *Distributed Computing*, 7(1):61–66, 1993. (Cited p. 187.)

[Her00] Ted Herman. Superstabilizing mutual exclusion. *Distributed Computing*, 13(1):1–17, 2000. (Cited p. 131.)

[HL98] Shing-Tsaan Huang and Tzong-Jye Liu. Four-state stabilizing phase clock for unidirectional rings of odd size. *Information Processing Letters*, 65(6):325–329, 1998. (Cited p. 131.)

[HNM99] Rodney R. Howell, Mikhail Nesterenko, and Masaaki Mizuno. Finite-state self-stabilizing protocols in message-passing systems. In *Proceedings of the 4th Workshop on Self-stabilizing Systems (WSS'99)*, pages 62–69, Austin, Texas, USA, June 5, 1999. (Cited p. 21.)

[Hua00] Shing-Tsaan Huang. The fuzzy philosophers. In *Proceedings of the International Parallel and Distributed Processing Symposium Parallel and Distributed Processing Workshops (IPDPS'00)*, pages 130–136, Cancun, Mexico, May 1-5, 2000. (Cited on pp. 170 and 171.)

[IJ90] Amos Israeli and Marc Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing (PODC'90)*, pages 119–131, Quebec City, Quebec, Canada, August 22-24, 1990. (Cited p. 19.)

[JADT02] Colette Johnen, Luc Onana Alima, Ajoy Kumar Datta, and Sébastien Tixeuil. Optimal snap-stabilizing neighborhood synchronizer in tree networks. *Parallel Processing Letters*, 12(3-4):327–340, 2002. (Cited p. 131.)

[Jou98] Yuh-Jzer Joung. Asynchronous group mutual exclusion (extended abstract). In *Proceedings of 17th Annual ACM Symposium on Principles of Distributed Computing (PODC'98)*, pages 51–60, Puerto Vallarta, Mexico, June 28 - July 2, 1998. (Cited p. 12.)

[KA97] Sandeep S. Kulkarni and Anish Arora. Multitolerant barrier synchronization. *Information Processing Letters*, 64(1):29–36, 1997. (Cited p. 132.)

[KIY13] Sayaka Kamei, Tomoko Izumi, and Yukiko Yamauchi. An asynchronous self-stabilizing approximation for the minimum connected dominating set with safe convergence in unit disk graphs. In *Proceedings of the 15th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'03)*, pages 251–265, Osaka, Japan, November 13-16, 2013. (Cited p. 130.)

[KK08] Sayaka Kamei and Hirotsugu Kakugawa. A self-stabilizing approximation for the minimum connected dominating set with safe convergence. In *Proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS'08)*, pages 496–511, Luxor, Egypt, December 15-18, 2008. (Cited p. 130.)

[KK12] Sayaka Kamei and Hirotsugu Kakugawa. A self-stabilizing 6-approximation for the minimum connected dominating set with safe convergence in unit disk graphs. *Theoretical Computer Science*, 428:80–90, 2012. (Cited p. 130.)

[KK13] Alex Kravchik and Shay Kutten. Time optimal synchronous self stabilizing spanning tree. In *Proceedings of the 27th International Symposium on Distributed Computing (DISC'13),*, pages 91–105, Jerusalem, Israel, October 14-18, 2013. (Cited p. 74.)

[KM06] Hirotsugu Kakugawa and Toshimitsu Masuzawa. A self-stabilizing minimal dominating set algorithm with safe convergence. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS'06)*, Rhodes Island, Greece, April 25-29, 2006. (Cited on pp. 18, 130, and 134.)

[KP93] Shmuel Katz and Kenneth J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7(1):17–26, 1993. (Cited p. 19.)

[KP99] Shay Kutten and Boaz Patt-Shamir. Stabilizing time-adaptive protocols. *Theoretical Computer Science*, 220(1):93–111, 1999. (Cited on pp. 19 and 128.)

[KPP+13] Shay Kutten, Gopal Pandurangan, David Peleg, Peter Robinson, and Amitabh Trehan. Sublinear bounds for randomized leader election. In *Proceedings of the 14th International Conference on Distributed Computing and Networking (ICDCN'13)*, pages 348–362, Mumbai, India, January 3-6, 2013. (Cited p. 40.)

[Kru56] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956. (Cited p. 13.)

[KUFM02] Yoshiaki Katayama, Eiichiro Ueda, Hideo Fujiwara, and Toshimitsu Masuzawa. A latency optimal superstabilizing mutual exclusion protocol in unidirectional rings. *Journal Parallel Distributed Computing*, 62(5):865–884, 2002. (Cited p. 131.)

[Lam74] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974. (Cited on pp. 12 and 170.)

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. (Cited p. 8.)

[Lan77] Gérard Le Lann. Distributed systems - towards a formal approach. In *Proceedings of IFIP Congress on Information Processing 77*, pages 155–160, Toronto, Canada, August 8-12, 1977. (Cited on pp. 12 and 40.)

[Lyn54] Roger C. Lyndon. On burnside's problem. *Transactions of the American Mathematical Society*, 77:202–215, 1954. (Cited p. 58.)

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996. (Cited on pp. 8, 34, 40, and 222.)

[Mis91] Jayadev Misra. Phase synchronization. *Information Processing Letters*, 38(2):101–105, 1991. (Cited on pp. 14 and 132.)

[Moo57] Edward F. Moore. The shortest path through a maze. In *Proceedings of an International Symposium on the Theory of Switching, Part II*, pages 285–292, April 2-5, 1957. (Cited p. 13.)

[MW87] Shlomo Moran and Yaron Wolfstahl. Extended impossibility results for asynchronous complete networks. *Information Processing Letters*, 26(3):145–151, 1987. (Cited on pp. 9 and 17.)

[NA02] Mikhail Nesterenko and Anish Arora. Stabilization-preserving atomicity refinement. *Journal of Parallel and Distributed Computing*, 62(5):766–791, 2002. (Cited on pp. 170 and 171.)

[NV01] Florent Nolot and Vincent Villain. Universal self-stabilizing phase clock protocol with bounded memory. In *Proceedings of the 20th IEEE International Conference on Performance, Computing, and Communications (IPCCC'01)*, pages 228–235, Phoenix, Arizona, USA, April 4-6, 2001. (Cited p. 131.)

[Pet82] Gary L. Peterson. An o(n log n) unidirectional algorithm for the circular extrema problem. *ACM Transactions on Programming Languages and Systems*, 4(4):758–762, 1982. (Cited on pp. 40 and 47.)

[Pri57] Robert C. Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957. (Cited p. 13.)

[Ray91] Michel Raynal. A distributed solution to the k-out of-m resources allocation problem. In *Advances in Computing and Information - Proceedings of the International Conference on Computing and Information (ICCI'91)*, pages 599–609, Ottawa, Canada, May 27-29, 1991. (Cited on pp. 12 and 170.)

[Seg83] Adrian Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, 29(1):23–34, 1983. (Cited on pp. 11 and 80.)

[Tel00] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2000. (Cited on pp. 8, 34, 35, and 222.)

[TJH10] Chi-Hung Tzeng, Jehn-Ruey Jiang, and Shing-Tsaan Huang. Size-independent self-stabilizing asynchronous phase synchronization in general graphs. *Journal of Information Science and Engineering*, 26(4):1307–1322, 2010. (Cited p. 131.)

[Var00] George Varghese. Self-stabilization by counter flushing. *SIAM Journal on Computing*, 30(2):486–510, 2000. (Cited on pp. 20 and 21.)

[XS06] Zhenyu Xu and Pradip K. Srimani. Self-stabilizing anonymous leader election in a tree. *International Journal of Foundations of Computer Science*, 17(2):323–336, 2006. (Cited p. 40.)

[YK89] Masafumi Yamashita and Tiko Kameda. Electing a leader when processor identity numbers are not distinct (extended abstract). In *Proceedings of the 3rd International Workshop on Distributed Algorithms (WDAG'89)*, pages 303–314, Nice, France, September 26-28, 1989. (Cited on pp. 15, 40, and 227.)

[YK96] Masafumi Yamashita and Tsunehiko Kameda. Computing on anonymous networks: Part i-characterizing the solvable cases. *IEEE Transactions on Parallel and Distributed Systems*, 7(1):69–89, 1996. (Cited on pp. 15 and 226.)

Annexe **A**

Résumé en français

Contents

Dans cette annexe, nous résumons les contributions et les perspectives de cette thèse. Dans un premier temps, nous détaillons le contexte dans lequel s'inscrit cette thèse en commençant par un exemple.

Lorsque Jane se réveille, des capteurs de mouvements détectent son réveil et allument les lampes progressivement ainsi que le chauffage de la salle de bain. Lors de son trajet jusqu'au travail, le GPS de Jane l'informe d'un accident signalé par d'autres utilisateurs. Elle peut alors changer son itinéraire afin d'éviter les bouchons générés par cet accident. Quand elle arrive au travail, elle trouve rapidement un place libre où se garer grâce aux capteurs qui surveillent l'occupation du parking. Pendant sa journée de travail, Jane échange des emails avec ses clients à l'autre bout du monde. Elle participe à une réunion par visioconférence avec une autre filiale et échange des données avec ses collègues via le réseau local de l'entreprise. Pendant qu'elle n'est pas à la maison, les capteurs de ses panneaux solaires détectent une grande production d'électricité à la mi-journée. Ils allument donc le chauffe-eau et le lave-vaisselle. Quand elle rentre, Jane vérifie les dernières informations sur Internet et partage les photos du weekend dernier avec sa famille grâce à un service de partage de fichiers sur le "cloud".

Chaque situation décrite dans l'exemple précédent utilise un système distribué. Ces systèmes sont omniprésents et inévitables dans notre vie de tous les jours. Avec de plus en plus d'utilisateurs, ces systèmes deviennent de plus en plus grands et complexes. Nous avons donc besoin d'algorithmes efficaces pour les faire fonctionner. De plus, les systèmes distribués sont très divers et peuvent être utilisés dans de nombreux contextes différents. Par exemple, les systèmes distribués peuvent être utilisés dans les maisons, les rues, les usines, comme présenté dans l'exemple précédent, mais aussi dans des contextes beaucoup plus hostiles (un réseau de capteurs sans-fils déployé dans un désert ou autour d'un volcan, par exemple). Cependant ces contextes peuvent être incertains, autrement dit le contexte n'est pas complètement connu au départ ou il est changeant. Par exemple, de larges systèmes composés de dispositifs bon marché et produits en masse sont très susceptibles d'être sujets à des dysfonctionnements ou des pannes. Ces dysfonctionnements et ces pannes sont imprévisibles, néanmoins le service fourni par le système doit toujours être disponible. La nature des systèmes elle-même peut être très dynamique comme par exemple dans les réseaux mobiles. En effet, un utilisateur de téléphone portable peut se déplacer et changer d'antenne-relais au milieu d'un appel, cet appel ne doit néanmoins pas être interrompu. Il faut donc que les systèmes distribués soient résistants aux incertitudes. Le développement de réseaux sociaux à grande échelle où des quantités gigantesques de données sont échangées de par le monde, entraîne un besoin de confidentialité de plus en plus important. Ce besoin montre que, dans certains cas, l'incertitude n'est pas un inconvénient mais plutôt une demande de l'utilisateur. La question de confidentialité a justifié le développement de solutions pour les réseaux anonymes. Un anonymat partiel peut aussi être obtenu dans les réseaux homonymes. Le besoin de confidentialité est généralement considéré comme un besoin de sécurité. Même si la sécurité n'est pas le sujet de cette thèse, nous étudions tout de même plusieurs niveaux d'anonymat dans nos solutions.

A.1 Contexte de la thèse

En informatique, un *système distribué* [Tel00, Lyn96] est un système informatique composé de plusieurs ordinateurs ou processus qui coopèrent pour atteindre un but commun. Plus précisément, un système distribué est un ensemble d'unités de calcul autonomes mais interconnectées. Une *unité de calcul* est un ordinateur, un cœur d'un processeur multi-cœur, un processus dans un système d'exploitation multitâches, *etc.* Pour simplifier, les ordinateurs, processeurs et processus seront nommés *processus* par la suite. Ces processus peuvent être distants géographiquement. *Autonome* signifie que chaque processus a son propre contrôle. Il ne dépend pas d'un contrôleur central. *Interconnecté* signifie que les processus sont capables d'échanger entre eux des informations, directement ou indirectement, notamment en envoyant des messages *via* des câbles ou des ondes radios, ou *via* des mémoires partagées. Cette définition inclut les ordinateurs parallèles, les réseaux d'ordinateurs, les réseaux de capteurs, les réseaux mobiles, les flottes de robots, *etc.*

A.1.1 Caractéristiques des systèmes distribués et différences avec les systèmes centraux.

Les systèmes distribués sont souvent définis par opposition aux systèmes centraux. En effet, les systèmes distribués ont des caractéristiques qui leurs sont propres :

- **Absence de temps global :** Dans les systèmes distribués, la vitesse de calcul de chaque processus est hétérogène et les communications sont généralement asynchrones. Les processus n'ont pas accès à une horloge globale. En particulier, leur horloge locale peut diverger.
- **Absence de connaissances globales :** Contrairement aux systèmes centraux où les décisions sont prises en fonction de l'état global du système, les processus d'un système distribué n'ont accès qu'à leurs connaissances locales, c'est-à-dire leur mémoire locale, pour décider de leur prochaine action. En particulier, même si la mémoire locale d'un processus peut être mise à jour lors de la réception d'informations, cette information peut être obsolète à cause de l'asynchronisme du système.
- **Non-déterminisme :** À cause de l'asynchronisme des processus et des communications, l'exécution d'un algorithme distribué déterministe peut mener à des résultats différents et le résultat obtenu n'est pas toujours prévisible. Au contraire, l'exécution d'un algorithme séquentiel déterministe ne dépend que de ses entrées.

A.1.2 Exemples de motivations et d'applications des systèmes distribués

Les systèmes distribués ont de nombreuses applications et sont omniprésents dans notre vie de tous les jours. Selon l'application, les systèmes distribués peuvent être tout simplement nécessaires ou peuvent être préférés aux systèmes séquentiels et centraux pour diverses raisons. Quelques exemples non-exhaustifs sont présentés ci-dessous.

Simplifier les communications. En 1969, un réseau étendu, nommé ARPANET, est créé entre de grandes universités américaines pour faciliter la coopération et l'échange de données entre ces organisations. ARPANET est l'ancêtre d'Internet qui connecte aujourd'hui des milliards d'ordinateurs et d'autres appareils.

De nos jours, nos communications dépendent grandement des systèmes distribués : emails, technologies de voix sur IP ou VoIP (par exemple, Skype, Google Talk, Discord), applications de messagerie instantanée (comme WhatsApp, Yahoo !Messenger, ou Google Hangouts), réseaux pair-à-pair (P2P) d'échange de fichiers (par exemple, Gnutella, eDonkey), *etc.*

Calculs plus rapides et à distance. En multipliant le nombre de processus, le calcul d'une tâche longue peut être partagée entre plusieurs processus et ainsi le calcul sera plus rapide. C'est l'objectif des ordinateurs parallèles. Par exemple, le superordinateur Deep Blue d'IBM a été conçu pour calculer rapidement des coups aux échecs. Des réseaux géographiquement étendus peuvent aussi être utilisés pour du calcul distribué. Par exemple, dans les projets de calcul volontaire, chacun peut offrir un peu de

la puissance de calcul ou de la mémoire de son ordinateur personnel pour aider au calcul d'une tâche difficile. Par exemple, le projet SETI@Home recherche des transmissions radio extraterrestres et le projet Rosetta@Home analyse la structure des protéines pour la recherche médicale.

Pour faciliter le calcul sur des réseaux distribués distants, beaucoup de compagnies proposent des services de "cloud", comme par exemple Amazon Elastic Compute Cloud ou Microsoft Azure. Le "cloud computing" propose un accès à la demande à une puissance de calcul et à du stockage distribués. Notez que certains services de "cloud" sont dédiés au stockage de fichiers, comme par exemple DropBox ou Google Drive.

Surveillance. Les *réseaux de capteurs sans fils* sont composés de nombreux capteurs générant des données à propos de leur environnement. Ces capteurs sont équipés de capacités de communication sans fil. Ces réseaux de capteurs peuvent être utilisés pour surveiller des catastrophes naturelles comme des éruptions volcaniques ou des séismes. Ils sont également de plus en plus utilisés dans les nouvelles technologies de domotique et de villes intelligentes pour surveiller la consommation en énergie, l'éclairage, *etc.* Les essaims de drones et les flottes de robots peuvent aussi être utilisées pour surveiller une zone et pour des applications militaires.

Améliorer la disponibilité et la résilience. En dupliquant une même tâche sur plusieurs processus, la disponibilité d'un service est améliorée face à la panne d'un processus. Notez que la réplication d'un calcul requiert un arbitrage entre les résultats des différents processus. Une technique similaire peut être utilisée pour améliorer la disponibilité des données en les copiant sur plusieurs disques de stockage. En particulier, la réplication de données peut être réalisée sur des serveurs de données géographiquement distants pour améliorer la résilience.

Mise en commun de ressources. Comme indiqué précédemment, les systèmes distribués permettent de partager des données, de la puissance de calcul, des disques de stockage, *etc.* Il est parfois nécessaire de partager d'autres périphériques, par exemple des imprimantes entre les employés d'une entreprise, car ces équipements sont coûteux.

A.1.3 Problèmes classiques des systèmes distribués

Les processus d'un système distribué cherchent à réaliser une tâche commune en utilisant leurs entrées locales. À cause des caractéristiques des systèmes distribués, la conception d'algorithmes demande de faire face à des problèmes fondamentaux afin de pouvoir résoudre des tâches de plus haut niveau. Quelques exemples sont listés ci-dessous.

- **Routage :** Un processus ne peut pas forcément communiquer directement avec n'importe quel autre processus. Lorsqu'un processus a besoin d'envoyer des informations à un autre, il le fait donc de manière indirecte. L'information passe de processus en processus jusqu'à atteindre sa destination. Ainsi, le but de certains problèmes est de déterminer par quel chemin doit passer l'information.

- **Accords :** En l'absence de contrôle central, les processus peuvent avoir besoin de décider et de se mettre d'accord sur certaines informations. C'est par exemple le cas dans les problèmes du consensus (binaire) et de l'élection de leader.

- **Allocation de ressources :** Quand des ressources sont partagées entre plusieurs processus, comme par exemple, une imprimante partagée entre les employés d'une entreprise, vous voulez vous assurer qu'un processus ayant besoin d'une ressource finit par y accéder (personne ne monopolise l'imprimante par exemple) et vous ne voulez pas qu'il y ait des conflits d'accès (par exemple, deux fichiers ne doivent pas s'imprimer en même temps). En général, le nombre de ressources partagées est beaucoup plus petit que le nombre de processus. Les problèmes d'allocations de ressources consistent à gérer un accès aux ressources qui soit équitable.

- **Construction de structures couvrantes :** La topologie d'un système distribué, c'est-à-dire les liens de communications entre processus, peuvent ne pas être organisés. Néanmoins, résoudre certains problèmes est plus simple et/ou plus rapide quand le système a une certaine structure, comme par exemple réaliser une diffusion depuis la racine d'un arbre. Par conséquent, construire une structure couvrante est un problème fondamental en algorithmique distribuée. Il peut s'agir par exemple de construire un arbre couvrant ou un clustering, c'est-à-dire des grappes de processus.

- **Coloriage :** Il est parfois nécessaire de différencier localement les processus. Il faut alors attribuer des couleurs aux processus selon certaines contraintes (par exemple, pas deux processus voisins de la même couleur) en utilisant un minimum de couleurs différentes.

- **Synchronisation :** Comme indiqué précédemment, les communications et les processus sont généralement asynchrones. Néanmoins, il est plus simple de concevoir des algorithmes pour des systèmes synchrones puisqu'il y a moins de non-déterminisme dans ces systèmes. De plus, il est impossible de résoudre certains problèmes sans hypothèses sur le synchronisme comme par exemple le consensus déterministe dans le cas où un processus peut tomber en panne [FLP85]. Par conséquent, certains problèmes s'intéressent à la synchronisation des processus.

Performances. La taille des systèmes distribués augmente avec la démocratisation des appareils connectés. Par exemple, le nombre d'utilisateurs d'Internet dans le monde est passé d'un milliard d'utilisateurs en 2005 (environ 16% de la population mondiale) à 3,5 milliards en 2016 (environ 47%). Avec l'essor des systèmes distribués, leur complexité augmente également. Par conséquent, pour que ces systèmes restent utilisables, il faut concevoir des algorithmes efficaces.

Premièrement, le calcul doit être rapide et le service fourni doit toujours être disponible. De plus, les systèmes distribués contiennent de plus en plus de systèmes embarqués, par exemple des capteurs sans fil, qui ont des ressources limitées (petite batterie, faible puissance de calcul, petite mémoire). Par conséquent, la complexité en mémoire, le nombre de messages échangés, et la complexité du calcul en lui-même doivent être faibles. Si ce n'est pas le cas, les processus risquent de ne même pas pouvoir exécuter leur algorithme ou de vider leur batterie.

A.1.4 Contexte incertain

Dans cette thèse, le contexte d'exécution du système distribué est dit incertain s'il n'est pas complètement connu au départ ou s'il est changeant. Nous nous intéressons à des systèmes qui ne sont pas complètement identifiés où des pannes peuvent se produire.

Dans le cas contraire, si aucune panne n'affecte le système, autrement dit le système satisfait continûment sa spécification, et si les processus sont identifiés, c'est-à-dire, ont un identifiant (ID) unique, la plupart des problèmes qui peuvent être résolus dans un système central, peuvent également être résolus dans un système distribué. Par exemple, pour calculer un arbre couvrant, les processus peuvent élire un *leader* (c'est-à-dire un unique processus distingué parmi les autres). Ce leader peut exécuter un snapshot pour collecter les états locaux, et en particulier les entrées du problème, de tous les autres processus. Par conséquent, le leader peut connaître toute la topologie et toutes les entrées du système, et il peut calculer le résultat (autrement dit, l'arbre) de façon centralisée, avant de l'envoyer à tous les autres processus.

Cette technique est très coûteuse et ne peut pas être appliquée dans des systèmes réels. En effet, elle nécessite une grande mémoire pour le leader, un grand nombre de messages échangés et un long temps de calcul. Bien évidemment, il existe des algorithmes beaucoup plus efficaces pour construire un arbre couvrant (par exemple, [DLV11a], [DLV11b] ou [ACD+16]), et une partie de la recherche en algorithmique distribuée se concentre sur la conception d'algorithmes efficaces sous ces conditions. Néanmoins, la technique présentée ci-dessus montre la solvabilité des problèmes dans les systèmes distribués.

Absence d'identification. À cause de la taille et de la complexité des systèmes distribués, supposer que les processus sont identifiés peut être irréaliste, en particulier pour des appareils bon marché et massivement produits. De plus, même si les processus sont identifiés, quelqu'un peut ne pas vouloir communiquer publiquement son ID pour des raisons de sécurité ou de confidentialité.

Cependant, dans un système *anonyme* où les processus n'ont pas d'IDs, beaucoup de problèmes fondamentaux deviennent impossible à résoudre. En particulier, il est impossible de casser les symétries de la topologie du réseau. Par exemple, le problème de l'élection de leader ne peut pas être résolu de manière déterministe dans un réseau anonyme puisque deux processus ne peuvent être distingués l'un de l'autre hormis par leurs entrées et leur degré (autrement dit, le nombre de processus avec qui ils peuvent communiquer directement). Yamashita et Kakugawa proposent un état de l'art des problèmes calculables dans les réseaux anonymes [YK96].

Pour contourner ces résultats d'impossibilité, il y a principalement deux approches. La première approche est la conception de solutions probabilistes. Par exemple, si deux processus voisins ne peuvent être distingués, ils peuvent "jeter une pièce" jusqu'à obtenir un résultat différent. Cependant, avec cette solution, la spécification du problème considéré est seulement assuré avec une certaine probabilité. D'autre part, la seconde approche consiste à considérer des modèles d'anonymat intermédiaires, ni complètement identifiés (où les processus ont un ID unique), ni complètement anonymes (où les processus n'ont pas d'ID). Par exemple, il est possible de considérer le modèle des processus

homonymes [YK89] dans lequel les processus ont des identifiants, mais ces identifiants ne sont pas nécessairement uniques. Dans ce cas, les processus ayant le même identifiant sont dits *homonymes*.

Présence de pannes. Quand la taille d'un système distribué augmente, il devient de plus en plus susceptible de subir la défaillance d'un processus. En effet, un processus peut tomber en panne, sa mémoire peut être corrompue, *etc.* De plus, les appareils composant les systèmes distribués sont souvent produits en masse et à coût réduit. Ils sont donc plus fragiles. Finalement, les communications sans fil sont de plus en plus utilisées alors qu'elles sont plus vulnérables. En 2016, le nombre d'objets connectés à Internet était estimé à 7 milliards. En ajoutant les ordinateurs, les smartphones et les tablettes, on atteint le nombre de 18 milliards d'appareils connectés. Dans des systèmes distribués de cette taille, il est impossible de supposer qu'aucune panne ne va se produire, ne serait-ce que pendant quelques heures.

Comme expliqué précédemment, les systèmes distribués sont omniprésents dans notre vie de tous les jours et les gens sont de plus en plus dépendants d'eux. Si une coupure de service, même temporaire, venait à toucher un tel système, les conséquences seraient importantes. Néanmoins, à cause de la complexité, de l'étendue et/ou de l'utilisation des systèmes distribués, assurer une maintenance humaine est souvent trop compliquée, trop lente ou même trop dangereuse. Par conséquent, les systèmes distribués doivent être résistants aux pannes. Nous détaillons les pannes considérées et étudions la tolérance aux pannes dans la section suivante.

A.1.5 Tolérance aux pannes

En informatique, une *panne* entraîne une *erreur* du système qui cause une *défaillance*. Un composant ou un système subit une défaillance lorsque son comportement n'est pas correct vis-à-vis de sa spécification. Une erreur est un état du système pouvant entraîner une défaillance. Il peut s'agir d'une *erreur logicielle* (par exemple, une division par zéro ou un pointeur non-initialisé) ou une *erreur physique* (par exemple, un câble débranché, une unité centrale éteinte, une coupure de connexion sans fil). Une panne est un évènement entraînant une erreur : soit une faute de programmation entraînant une erreur logicielle, soit un évènement physique (par exemple, une coupure de courant ou des perturbations dans l'environnement du système) entraînant une erreur physique. Dans cette thèse, nous considérons seulement des erreurs physiques.

Classification des pannes. Les pannes peuvent être classées selon

- leur localisation : si le composant touché par la panne est un lien de communication ou un processus ;
- leur origine : si la faute est *bénigne* (due à un problème physique) ou *maligne* (due à une malveillance) ;
- leur durée : si la faute est *permanente* (plus longue que la durée restante du calcul), *transitoire* ou *intermittente*. Il y a une légère différence entre les pannes transitoires et les pannes intermittentes. En moyenne, durant une exécution, une panne transi-

toire n'affecte le système qu'une seule fois, alors qu'une panne intermittente affecte le système plusieurs fois.

- leur détection : selon si les processus peuvent détecter ou non en fonction de leur état local s'ils sont touchés par une panne.

Quelques exemples de pannes : la panne définitive d'un processus (c'est-à-dire un processus qui stoppe l'exécution de son algorithme), un processus Byzantin (autrement dit, qui a un comportement arbitraire) ou une panne transitoire (c'est-à-dire un composant qui a temporairement un comportement incorrect, mais cette panne ne provoque pas de dommage permanent au matériel).

Algorithmes robustes *vs.* algorithmes stabilisants. Deux approches principales ont été étudiées pour concevoir des systèmes distribués résistants aux pannes : une approche pessimiste (la conception d'algorithmes robustes) et une approche optimiste (la conception d'algorithmes stabilisants). (Notez qu'il existe des algorithmes à la fois robustes et stabilisants, voir par exemple [GP93].) Dans un algorithme *robuste*, toute information reçue est suspectée afin de garantir un comportement correct des processus qui ne sont pas défaillants. Des stratégies sont utilisées dans ces algorithmes comme le vote permettant de considérer des informations uniquement si un nombre suffisant d'autres processus déclarent avoir reçu une information similaire. Par conséquent, les algorithmes robustes permettent de résister aux pannes permanentes et doivent être considérés quand une interruption de service, même temporaire, est inacceptable. Au contraire, quand des interruptions de service à la fois courtes et rares peuvent être acceptées (courtes et rares comparées à la disponibilité globale du service), les algorithmes stabilisants offrent une approche plus légère pour résister aux pannes transitoires. Un exemple de cette approche est l'autostabilisation.

L'*autostabilisation* [Dij74, Dij86] est une approche polyvalente qui permet aux systèmes distribués de résister aux pannes transitoires. Après la fin des pannes transitoires, les processus, même ceux n'ayant pas été frappés par une panne, peuvent avoir un comportement incorrect. Si le système est autostabilisant, il retrouve en temps fini un comportement correct, sans aucune aide extérieure (en particulier, sans intervention humaine). Notez que la récupération ne dépend ni de la nature (c'est-à-dire, si les pannes affectent des processus et/ou des liens de communications) ni de leur étendue (c'est-à-dire, combien de composants sont touchés). Seules les modifications du code des processus sont exclues. La polyvalence de l'autostabilisation a deux principaux inconvénients. Premièrement, la spécification du système n'est pas assurée pendant la convergence, autrement dit il n'y a pas de garanties de sûreté pendant celle-ci. De plus, les processus ne sont pas capables de détecter localement la fin de la convergence. Il est donc impossible d'assurer une détection de terminaison. Pour contrer les inconvénients de l'autostabilisation, plusieurs variantes ont été proposées, comme par exemple, la stabilisation instantanée [BDPV07] ou la superstabilisation [DH97].

A.2 Contributions

Dans cette thèse, nous étudions des problèmes classiques de l'algorithmique distribuée dans des contextes incertains. Par incertain, nous désignons des contextes d'exécution de

systèmes distribués qui ne sont pas complètement connus au départ ou qui sont changeants. Nous nous focalisons sur deux types d'incertitudes : les processus qui ne sont pas complètement identifiés et la présence de pannes. Plus précisément, nous explorons deux axes de recherche :

- aller vers plus d'anonymat en proposant des algorithmes déterministes pour les réseaux de processus homonymes et anonymes,
- assurer plus de garanties de sûreté pendant la convergence d'algorithmes déterministes autostabilisants.

tout en proposant des solutions efficaces.

Après avoir introduit le contexte et un état de l'art élargi dans le chapitre 1, le chapitre 2 présente les modèles de calcul sur lesquels se base l'ensemble des contributions. Nous détaillons dans la suite de ce résumé les contributions (sections A.2.1-A.2.4) correspondant aux chapitres 3 à 6, avant de conclure par des perspectives en section A.3 (chapitre 7).

A.2.1 Élection de leader dans des anneaux unidirectionnels de processus homonymes

La première contribution de cette thèse se focalise sur l'élection de leader dans des anneaux statiques unidirectionnels contenant des processus homonymes, autrement dit les processus sont identifiés mais plusieurs processus peuvent avoir le même ID, appelé étiquette dans ce contexte. Nous utilisons ici le modèle à passage de messages.

Nous prouvons que l'élection de leader avec terminaison implicite est impossible à résoudre dans un anneau unidirectionnel dont l'étiquetage est symétrique. Par conséquent, nous considérons les anneaux dont l'étiquetage est asymétrique. Nous montrons qu'il est impossible de résoudre l'élection de leader avec terminaison explicite dans la classe $\mathcal{U}^*$ qui contient tous les anneaux unidirectionnels avec au moins une unique étiquette. Par conséquent, il est impossible de résoudre l'élection de leader avec terminaison explicite dans la classe $\mathcal{A}$ de tous les anneaux unidirectionnels avec un étiquetage asymétrique. Nous prouvons également qu'il est impossible de résoudre l'élection de leader avec terminaison implicite dans la classe $\mathcal{K}_k$ qui contient les anneaux unidirectionnels ne contenant pas plus de $k \geq 1$ processus ayant la même étiquette. En effet, $\mathcal{K}_k$ contient des anneaux symétriques.

Par la suite, nous proposons trois algorithmes U_k, A_k et B_k. L'algorithme U_k est un algorithme d'élection de leader avec terminaison explicite pour la classe $\mathcal{U}^* \cap \mathcal{K}_k$, pour tout $k \geq 1$. Il est asymptotiquement optimal en temps avec $\Theta(kn)$ unités de temps et en mémoire puisqu'il nécessite $O(\log k + b)$ bits par processus, où b est le nombre de bits nécessaires pour stocker une étiquette. Sa complexité en message est $O(kn)$. Les algorithmes A_k et B_k résolvent tous les deux l'élection de leader avec terminaison explicite pour la classe $\mathcal{A} \cap \mathcal{K}_k$, pour tout $k \geq 1$. A_k est asymptotiquement optimal en temps ($\Theta(kn)$ unités de temps) mais requiert $O(knb)$ bits par processus et $O(kn^2)$ envois de messages. Au contraire, B_k est asymptotiquement optimal en mémoire ($O(\log k + b)$ bits par processus) mais sa complexité en temps est $O(k^2n^2)$ et sa complexité en message

est $O(k^2n^2)$.

A.2.2 Élection de leader autostabilisante sous démon inéquitable

Tout comme dans le chapitre 3, nous considérons dans le chapitre 4 le problème de l'élection de leader, mais dans un contexte différent. Nous proposons un algorithme d'élection de leader autostabilisant silencieux, nommé $\mathcal{LE}$, pour tout réseau statique et identifié de topologie arbitraire, connectée et bidirectionnelle. L'algorithme $\mathcal{LE}$ est écrit dans le modèle à états, ne requiert aucune connaissance globale sur le réseau (par exemple, aucune borne supérieure sur le nombre de processus ou le diamètre) et suppose le démon distribué inéquitable.

Depuis une configuration arbitraire, $\mathcal{LE}$ converge vers une configuration terminale en au plus $3n + \mathcal{D}$ rondes, où $\mathcal{D}$ est le diamètre du réseau, et nous exhibons un réseau pour tout $n \geq 4$ et tout $\mathcal{D}$ tel que $2 \leq \mathcal{D} \leq n - 2$ dans lequel il existe une exécution durant exactement $3n + \mathcal{D}$ rondes. Dans cette configuration terminale, tous les processus connaissent l'ID du leader et un arbre couvrant enraciné au leader est défini. $\mathcal{LE}$ est asymptotiquement optimal en mémoire avec $\Theta(\log n)$ bits par processus.

Nous montrons que $\mathcal{LE}$ stabilise en un nombre polynomial de pas de calcul. En effet, il converge en $\Theta(n^3)$ pas. Nous étudions la complexité en pas de calcul des algorithmes précédents ayant les meilleurs performance sous les mêmes conditions, c'est-à-dire, sans connaissance globale exigée et prouvés sous démon distribué inéquitable. Pour tout $n \geq 5$, nous prouvons qu'il existe un réseau dans lequel il y a une exécution de l'algorithme proposé dans [DLV11a], noté ici $\mathcal{DLV}_1$, qui stabilise en $\Omega(2^{\lfloor \frac{n-1}{4} \rfloor})$ pas. De même, nous prouvons que pour un $\alpha \geq 3$ donné, pour tout $\beta \geq 2$, il existe un réseau de $n = 2^\alpha \times \beta$ processus dans lequel il y a une exécution possible de l'algorithme proposé dans [DLV11b], noté ici $\mathcal{DLV}_2$, qui stabilise en $\Omega(n^{\alpha+1})$. Par conséquent, les temps de stabilisation de $\mathcal{DLV}_1$ and $\mathcal{DLV}_2$ en nombre de pas de calcul ne sont pas polynomiaux.

A.2.3 Stabilisation progressive sous (τ, ρ)-dynamicité et unisson

Dans le chapitre 5, nous proposons une variante de l'autostabilisation, nommée stabilisation progressive sous (τ, ρ)-dynamicité. Cette variante est spécialement conçue pour les réseaux dynamiques. En effet, un algorithme est progressivement stabilisant sous (τ, ρ)-dynamicité s'il est autostabilisant et s'il satisfait les propriétés supplémentaires suivantes. Après au plus τ pas dynamiques de type ρ en partant d'une configuration légitime, un algorithme progressivement stabilisant retrouve tout d'abord très rapidement une configuration depuis laquelle une garantie de sûreté minimum est assurée. Ensuite, il converge progressivement vers des spécifications offrant une qualité de service de plus en plus importante, jusqu'à retrouver une configuration depuis laquelle les deux conditions suivantes sont vérifiées. Sa spécification initiale est satisfaite et, si au plus τ pas ρ-dynamiques affectent à nouveau le système, il est prêt à réaliser une convergence progressive.

Nous illustrons cette nouvelle propriété en proposant un algorithme progressivement

stabilisant, noté $\mathcal{DSU}$, pour le problème de l'unisson. $\mathcal{DSU}$ est conçu dans le modèle à états pour tout réseau anonyme de topologie arbitraire initialement connectée et il suppose le démon distribué inéquitable. Il est progressivement stabilisant sous $(1, \mathsf{BULCC})$-dynamicité. Un pas BULCC-dynamique contient un nombre fini mais non-borné de changements topologiques tel que, après un tel pas, le réseau : (1) contient au plus N processus (N est une borne supérieure sur le nombre de processus dans le système à tout moment), (2) est connecté et (3) si la période des horloges α est strictement plus grande que 3, tous les processus ayant rejoint le système doivent être liés à au moins un processus qui était déjà dans le système avant le pas dynamique (sauf si tous ces processus ont quitté le système). (Nous étudions la nécessité de ces conditions.) En partant d'une configuration satisfaisant l'unisson fort (il y a au plus deux valeurs différentes d'horloges et ces valeurs sont consécutives), si un pas BULCC-dynamique affecte le système, $\mathcal{DSU}$ satisfait immédiatement l'unisson partiel (les horloges de deux processus voisins diffèrent d'au plus un incrément, sauf pour les processus entrants). Puis, en une ronde, il satisfait l'unisson faible (les horloges de tous processus voisins diffèrent d'au plus un incrément) et converge vers l'unisson fort en $(\mu + 1)\mathcal{D}_1 + 2$ rondes, où μ est un paramètre supérieur ou égal à $\max(2, N)$ et $\mathcal{D}_1$ est le diamètre du réseau après le pas dynamique.

A.2.4 Concurrence et allocation de ressources

Finalement, dans le chapitre 6, nous proposons une propriété appelée concurrence maximale pour caractériser le niveau de concurrence qui peut être réalisé dans les problèmes d'allocation de ressources. Cette notion généralise des notions similaires préalablement définies pour des problèmes spécifiques, par exemple, ℓ-interblocage [FLBB79] définie pour la ℓ-exclusion et la (k, ℓ)-vivacité [DHV03] définie pour la k-parmi-ℓ exclusion.

Nous montrons que, même si la concurrence maximale peut être réalisée dans certains problèmes comme la ℓ-exclusion [FLBB79], il est impossible de l'atteindre dans une grande classe de problèmes d'allocation de ressources nommée allocation de ressources locales (LRA). Néanmoins, nous prouvons que la concurrence forte, un niveau de concurrence haut mais pas maximal, peut être réalisé dans le problème de la LRA. Plus précisément, nous proposons un algorithme fortement concurrent et instantanément stabilisant de LRA, nommé $\mathcal{LRA} \circ \mathcal{TC}$, pour les réseaux connectés bidirectionnels de topologie quelconque. $\mathcal{LRA} \circ \mathcal{TC}$ est écrit dans le modèle à états et suppose un démon faiblement équitable.

A.3 Perspectives

Les perspectives de cette thèse comportent trois axes de recherche principaux.

Homonymes et autostabilisation. Le modèle des processus homonymes a été très peu étudié. En particulier, à notre connaissance, aucun algorithme autostabilisant n'a été proposé pour les réseaux contenant des processus homonymes. Cependant, nos résultats du chapitre 3, en particulier l'algorithme A_k, semblent très prometteur pour un passage à l'autostabilisation. Ainsi, une perspective de cette thèse est la conception d'algorithmes autostabilisants pour les réseaux homonymes, tout d'abord pour l'élection de leader, puis pour d'autres problèmes.

Stabilisation progressive. Dans le chapitre 5, nous proposons une nouvelle propriété pour les réseaux dynamiques, la stabilisation progressive sous (τ, ρ)-dynamicité et nous illustrons cette propriété pour $\tau = 1$ avec un algorithme d'unisson. La généralisation pour $\tau > 1$ reste une question ouverte. De plus, réaliser cette propriété piur d'autres problèmes (dynamiques) est une extension naturelle qui pourrait mener à une meilleure compréhension et donc permettre la généralisation de l'approche par la conception d'un transformateur (c'est-à-dire un algorithme qui procurerait la propriété de stabilisation progressive à des algorithmes simplement autostabilisants).

Concurrence. Comme indiqué dans le chapitre 6, la question de la concurrence dans les problèmes d'allocation de ressources a été peu étudiée. Pourtant, elle est fondamentale pour maximiser l'utilisation des ressources et minimiser le temps d'attente des processus demandeurs. Il a été prouvé que le niveau maximal de concurrence (appelé ici concurrence maximale) ne peut être réalisé dans de nombreux problèmes, plus précisément en k-parmi-ℓ exclusion, en coordination de comités et en allocation de ressources locales (voir chapitre 6). Les seuls problèmes pour lesquels nous savons que la concurrence maximale est réalisable sont la ℓ-exclusion [FLBB79] et trivialement l'exclusion mutuelle. Par conséquent, le niveau de concurrence qui peut être obtenu dans d'autres problèmes d'allocation de ressources, l'exclusion mutuelle de groupe ou le problème des philosophes qui boivent, mérite d'être étudié.

Abstract

Distributed systems become increasingly wide and complex, while their usage extends to various domains (*e.g.*, communication, home automation, monitoring, cloud computing). Thus, distributed systems are executed in diverse contexts. In this thesis, we focus on uncertain contexts, *i.e.*, the context is not completely known *a priori* or is unsettled. More precisely, we consider two main kinds of uncertainty: processes that are not completely identified and the presence of faults. The absence of identification is frequent in large networks composed of massively produced and deployed devices. In addition, anonymity is often required for security and privacy. Similarly, large networks are exposed to faults (*e.g.*, process crashes, wireless connection drop), but the service must remain available.

This thesis is composed of four main contributions. First, we study the leader election problem in unidirectional rings of homonym processes, *i.e.*, processes are identified but their ID is not necessarily unique. Then, we propose a silent self-stabilizing leader election algorithm for arbitrary connected network. This is the first algorithm under such conditions that stabilizes in a polynomial number of steps. The third contribution is a new stabilizing property designed for dynamic networks that ensures fast and gradual convergences after topological changes. We illustrate this property with a clock synchronizing algorithm. Finally, we consider the issue of concurrency in resource allocation problems. In particular, we study the level of concurrency that can be achieved in a wide class of resource allocation problem, *i.e.*, the local resource allocation.

Keywords. Distributed algorithms, fault-tolerance, self-stabilization, anonymity, dynamic networks.

Résumé

Les systèmes distribués sont de plus en plus grands et complexes, alors que leur utilisation s'étend à de nombreux domaines (par exemple, les communications, la domotique, la surveillance, le "cloud"). Par conséquent, les contextes d'exécution des systèmes distribués sont très divers. Dans cette thèse, nous nous focalisons sur des contextes incertains, autrement dit, le contexte n'est pas complètement connu au départ ou il est changeant. Plus précisément, nous nous focalisons sur deux principaux types d'incertitudes : une identification incomplète des processus et la présence de fautes. L'absence d'identification est fréquente dans de grands réseaux composés d'appareils produits et déployés en masse. De plus, l'anonymat est souvent une demande pour la sécurité et la confidentialité. De la même façon, les grands réseaux sont exposés aux pannes comme la panne définitive d'un processus ou une perte de connexion sans fil. Néanmoins, le service fourni doit rester disponible.

Cette thèse est composée de quatre contributions principales. Premièrement, nous étudions le problème de l'élection de leader dans les anneaux unidirectionnels de processus homonymes (les processus sont identifiés mais leur ID n'est pas forcément unique). Par la suite, nous proposons un algorithme d'élection de leader silencieux et autostabilisant pour tout réseau connecté. Il s'agit du premier algorithme fonctionnant sous de telles conditions qui stabilise en un nombre polynomial de pas de calcul. La troisième contribution est une nouvelle propriété de stabilisation conçue pour les réseaux dynamiques qui garantit des convergences rapides et progressives après des changements topologiques. Nous illustrons cette propriété avec un algorithme de synchronisation d'horloges. Finalement, nous considérons la question de la concurrence dans les problèmes d'allocation de ressources. En particulier, nous étudions le niveau de concurrence qui peut être atteint dans une grande classe de problèmes d'allocation de ressources, l'allocation de ressources locales.

Mots-clés. Algorithmes distribués, tolérance aux pannes, autostabilisation, anonymat, réseaux dynamiques.