Faculty II – Computing Science, Business Administration, Economics, and Law
Department of Computing Science

Recurrence in Self-Stabilization

Theory, Verification, and Application

This thesis is submitted in partial fulfillment of the requirements for the Degree of

Doctor rerum naturalium (Dr. rer. nat.)

in Computer Science from the Carl von Ossietzky University of Oldenburg

By

Oday Jubran

Born in Jerusalem on May 05, 1986

First Referee
Prof. Dr.-Ing. Oliver Theel
Carl von Ossietzky University of Oldenburg

Second Referee
Prof. Dr. Volker Turau
Hamburg University of Technology

Disputation Date: November 18, 2016

Declaration

I hereby certify that I am the sole author of my thesis, and that I have used only the listed resources.

Erklärung

Hiermit versichere ich, dass ich die Arbeit selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Oday Jubran

Abstract

When a system turns out to be exposed to faults, fault tolerance becomes necessary to guarantee continuous and useful behavior of the system. Self-stabilization is a fault tolerance concept which ensures that the system finally recovers itself from failures, due to transient faults, without voluntarily running into such. In distributed systems, self-stabilization is of a particular interest, as a system's component usually does not have full knowledge about the configuration (global state) and nevertheless, the reaction of the component has to direct the system's behavior towards preserving the system's desired properties. In the past 40 years, related work considered the design of self-stabilizing distributed algorithms with low cost, in the sense that an algorithm recovers quickly to a safe behavior, uses low space, and works under several topologies and schedulers.

In this work, the self-stabilization concept is generalized to pertain a sort of properties, other than the classical ones. In this concept, a property is based on a measure, that is defined as the ratio of configurations that satisfy some condition in each execution of a system, which is signified as the *recurrence* of the condition in the execution. Self-stabilization with respect to this property implies the convergence of a system to an execution suffix that guarantees a minimum recurrence of a condition. This generalized concept of self-stabilization provides the ability to consider the convergence of a system to reach its highest performance or quality of service. To this end, this work concerns designing, analyzing, and re-engineering self-stabilizing systems, that solve common problems in distributed computing, to be efficient wrt. recurrence properties. First, the mutual exclusion problem is tackled by presenting a self-stabilizing mutual exclusion algorithm that has optimal stabilization time, and achieves optimal service time under the synchronous scheduler. Second, the problem of educated unique process selection is introduced. This problem concerns selecting processes to be granted a privilege based on local or global criteria, with a special consideration of the fairness property. With this consideration, self-stabilizing algorithms, that are based on token passing, can be further optimized while preserving fairness under reasonable conditions. Two algorithms providing educated and fast unique process selection are presented. Third, the Time-Division-Multiple-Access (TDMA) slot assignment problem is considered for networks having tree topology, where the components communicate through a limited bandwidth. A formal analysis of the scheduling efficiency in terms of the cost of clock synchronization, and optimizing the length of guard intervals is provided. Finally, an automatic verification approach for recurrence properties over infinite executions is presented. The approach reduces the problem of verifying recurrence properties over infinite executions to finding counterexamples with a given fixed length.

Zusammenfassung

Wenn ein System Fehlern ausgesetzt ist, dann ist Fehlertoleranz unerlässlich, um kontinuierliches und nützliches Verhalten des Systems zu garantieren. Selbststabilisierung ist ein Konzept der Fehlertoleranz, welches eine Wiederherstellung des Systems von Ausfällen durch vorübergehende Fehler garantiert, ohne von sich aus in einen solchen zu geraten. In verteilten Systemen ist Selbststabilisierung von besonderem Interesse, da normalerweise keine Komponente des Systems völlige Kenntnis über die Konfiguration (den globalen Zustand) des Systems hat. Trotzdem muss die Reaktion einer Komponente das Systemverhalten so beeinflussen, dass die gewünschten Eigenschaften des Systems bewahrt werden. In den letzten 40 Jahren wurde in der Literatur das Design von effizienten selbststabilisierenden verteilten Algorithmen mit geringen Kosten betrachtet, das heißt, dass das System schnell stabilisiert, wenig Speicher verbraucht und auf mehrere Topologien und Scheduler anwendbar ist.

In dieser Arbeit wird das Konzept der Selbststabilisierung generalisiert, um mehrere Eigenschaftsarten abzudecken. In diesem Konzept basieren die Eigenschaften auf einem Maß, welches als das Verhältnis der Konfigurationen definiert ist, die eine bestimmte Bedingung in jeder Systemausführung erfüllen zu allen möglichen Konfigurationen. Dieses Verhältnis wird als *Rekurrenz* der Bedingung in der Ausführung bezeichnet. Selbststabilisierung bezüglich dieser Eigenschaft impliziert die Konvergenz eines Systems zu einem Ausführungssuffix, der eine minimale Rekurrenz der Bedingung garantiert. Dieses generalisierte Konzept ermöglicht die Betrachtung der Konvergenz eines Systems hin zu seiner maximalen Leistungsfähigkeit oder zu höchstmöglichen Servicequalität. Hierzu wird das Design, die Analyse und das Re-engineering von selbststabilisierenden Systemen, die bekannte Probleme in verteilten Systemen lösen, angewendet, um effizient bezüglich der Rekurrenzeigenschaften zu sein. Als Erstes wird das Problem des wechselseitigen Ausschlusses (engl. Mutual Exclusion) betrachtet. Es wird ein selbststabilisierender Algorithmus präsentiert, der eine optimale Stabilisierungszeit und optimale *Service-Time* unter einem synchronen Scheduler besitzt. Als Zweites wird das Problem der *Educated-Unique-Process-Selection* eingeführt. Dieses Problem befasst sich mit der Auswahl eines Prozesses, um ihm ein Privileg, basierend auf lokalen oder globalen Kriterien und unter einer spezifischen Betrachtung der Fairness-Eigenschaft, zu gewähren. Hierzu können selbststabilisierende Algorithmen basierend auf Tokenweitergabe noch weiter optimiert werden, wobei Fairness, unter bestimmten und vernünftigen Konditionen, erhalten bleibt. Es werden zwei selbststabilisierende Algorithmen für dieses Problem präsentiert. Als Drittes wird das Problem der Slot-Zuteilung in zeitbasierten Multiplexverfahren in Sensornetzen mit Baumtopologien und begrenzten Bandbreiten betrachtet. Es wird eine formale Analyse der Effizienz der Zeitplanung bezüglich der Taktsynchronisierung und die Optimierung der Guard-Intervalle vorgestellt. Zum Schluss wird eine automatische Verifikationsmethode für Rekurrenzeigenschaften auf unendlichen Ausführungen präsentiert. Diese Methode reduziert das Problem der Verifikation von Rekurrenzeigenschaften für unendliche Ausführungen auf das Nachweisen von Gegenbeispielen einer gegebenen festen Längen.

Acknowledgement

"It is impossible to feel grateful and depressed in the same moment" – Naomi Williams. It is my great pleasure to recognize all who have been involved to bring me to this stage. Foremost, my gratitude goes to who have direct impact on my doctoral study.

Prof. Dr.-Ing. Oliver Theel, Supervisor and First Referee
: I highly appreciate the distinguished work and study environment he has dedicated. My gratefulness goes to his numerous support, flexibility, kindness, patience, and continued guidance throughout this journey. Sincere thanks for being always there, especially, in critical situations.

(Dr.-Ing. to be soon) Eike Möhlmann, Office Mate and Co-worker
: I am incredibly grateful for his valuable support at all levels, especially, for embedding me in the University's environment and the surrounding culture. I highly appreciate our discussions and brainstorming sessions.

Dr. Bernd Westphal, Co-author and Former Supervisor
: The superior effect of his involvement is always recognizable. His influential guidance within my graduate studies is praised.

The examination board members, beside Professor Theel

Prof. Dr. Ernst-Rüdiger Olderog, Chairperson

Prof. Dr. Volker Turau, Second Referee

Dr. Astrid Rakow, Board Member
: Special thanks for her valuable contribution in the last moments.

The members of the System Software and Distributed Systems Group

In addition, I would like to acknowledge the organizations, that granted me funding for my graduate studies.

DFG: German Research Foundation
: It has funded a full research position during my doctoral study.

DAAD: German Academic Exchange Service
: It has funded a full scholarship for my Master's study.

I still owe sincere thanks to Ms. Ursula Epe, the Computer Science program coordinator at the University of Freiburg. Her urgent contribution at the very beginning of my graduate studies is unforgettable.

A special appreciation goes to the staff and members of Bethlehem University for dedicating an amiable and social atmosphere in my undergraduate education.

Last but not least, my deepest gratitude goes to my family and friends for their endless support and encouragement.

Contents

1 Introduction **1**
1.1 Fault Tolerance in Distributed Systems 1
1.2 Self-Stabilization 3
1.3 General Problem Statement 5
1.4 General Contribution 6
1.5 Outline 7

2 Preliminaries and Notation **9**
2.1 Basics 9
2.2 Topology 10
2.3 Discrete Systems 12
2.4 Real-Time Systems 15

3 Recurrence in Self-Stabilization **17**
3.1 Beyond Safety in Self-Stabilization 17
3.2 Recurrence in Discrete Systems 19
3.3 Recurrence in Real-Time Systems 23
3.4 Remarks 24

4 Mutual Exclusion **27**
4.1 The Mutual Exclusion Problem 27
4.2 Problem Statement and Assumptions 30
4.3 Mutual Exclusion Algorithms 31
4.4 Correctness and Time Complexity 36
4.5 Convergence Time Optimality under the Synchronous Scheduler 44
4.6 Remarks 45

5 Fast and Educated Unique Process Selection **47**
5.1 The Unique Process Selection Problem 47
5.2 Problem Statement and Assumptions 49
5.3 PIF: Propagation of Information with Feedback 51
5.4 Exploiting Synchronicity for Immediate Feedback 55
5.5 Educated Unique Process Selection 58
5.6 Correctness and Time Complexity 63
5.7 Remarks 72

6 TDMA Slot Assignment **75**
6.1 The TDMA Slot Assignment Problem 75
6.2 Problem Statement and Assumptions 79

6.3 System Model . . . 81
6.4 Worst and Best Slot Assignments . . . 86
6.5 Guard Time Optimization . . . 90
6.6 Recurrence Property . . . 95
6.7 Case Study: A Wireless Fire Alarm System . . . 98
6.8 Remarks . . . 107

7 Automatic Verification of Recurrence Properties **109**
7.1 Automatic Verification of Self-Stabilizing Systems . . . 109
7.2 Problem Statement and Assumptions . . . 111
7.3 Counterexamples of Fixed Length . . . 111
7.4 Model Checking Recurrence Properties . . . 115
7.5 Case Study: A Mutual Exclusion Algorithm . . . 117

8 Conclusion **121**
8.1 Summary . . . 121
8.2 Future Prospects . . . 123

Appendix A **125**

Appendix B **135**

List of Figures **141**

List of Tables **143**

List of Algorithms **145**

Bibliography **147**

Author's Publications **159**

Author's Biography **161**

Index **163**

1 Introduction

"I thought that in [Dij74] I had published three solutions, but later I learned that I had also published three problems, as the programs had been given without a demonstration of their correctness" – Dijkstra [Dij86]. This statement portrays how difficult it was, to find correctness proof for the algorithms presented by Dijkstra in his seminal work [Dij74], along 12 years. The algorithms – at that time – addressed systems having multiple processes that do not share a common global memory. The aim of the algorithms is to provide ability to the processes to achieve a global agreement or some sort of synchronization between them without the need of a shared memory or even an initial setting. Thus, both [Dij74, Dij86] provide a positive answer to the question, whether it is possible for participating components in a distributed system to agree on some specification, without the existence of a global control or an initial setting.

This thesis addresses a particular concept of fault tolerance in distributed systems. In this concept, a system is required to recover itself from failures due to transient, temporary, infrequent faults or dynamic topology changes, and to be able to finally execute correctly starting from any initial configuration – or global state – given enough time for recovery. This particular concept is called *self-stabilization*. The thesis contributes in generalizing the concept of self-stabilization to cover some sort of properties other than the classical ones. It applies the generalized concept to solve and extend well-known problems in distributed computing, and presents an automatic verification approach to verify properties of the generalized concept.

This chapter provides an introduction and motivation to this work. Section 1.1 provides an overview about fault tolerance in distributed systems. Next, Section 1.2 presents the concept of self-stabilization. After that, Section 1.3 demonstrates the general problem statement of this work. Section 1.4 states the general contribution. Finally, Section 1.5 presents the outline of this thesis.

1.1 Fault Tolerance in Distributed Systems

A distributed system can be viewed as a collection of components that operate independently, communicate with each other by passing messages, and appear to the users as a single system [TVS07]. The typical abstract model of a distributed system is a graph of vertices linked with edges, where each vertex can communicate with some vertices, namely its neighbors.

Distributed systems are found nowadays in numerous applications and industrial areas. Some examples are as follows: a system with multiple processes that do not share a common memory, a computer network with multiple diverse machines, the Internet, a wireless sensor network, a database system with multiple storage media, and many others. Some of the purposes of distributed systems are to distribute tasks among many

components towards reducing the burden on each component, to enable multiple users at the same time, and to cover large geographical areas.

Distributed systems are usually exposed to many kinds of faults. As examples, memory allocation problems may affect the performance of systems. Noises may interfere radio signals and disturb communication in wireless sensor networks. Adding or removing components in a network may affect the topological, routing, and scheduling schemes. In addition, attacks may disrupt a system's security. From this point, distributed systems are supposed to be able to react against faults.

The challenge of overcoming runtime and dynamic changes of the topology, deviation from correct execution due to faults, and renouncing external initialization in distributed systems lies in the fact that each component may not be able to reach all other components with one message or broadcast. This happens in many situations. For example, in wireless sensor networks, there might exist large distances between the components, such that one broadcast cannot reach all of them. In wired networks, there might be diverse machines that develop some routing scheme for message delivery. Thus, in many cases, a distributed system cannot recover itself easily as a non-distributed one.

Fault Tolerance

Fault tolerance concerns the ability of a system to continue functioning in the presence of faults, and to which degree the system can do it [TVS07, Gär99]. The methodology of applying fault tolerance to a system depends on many parameters of the system and the system's environment, such as the kind and duration of faults, and the consequences of faults.

Fault tolerance considers the system's reaction against faults, and therefore, it impacts the notion of *dependability* of a system [TVS07]. Dependability is based on many criteria. First, *availability* is a metric that indicates the probability at which a system provides "useful output." Second, *reliability* indicates whether a system can continue running correctly for some amount of time without a failure. Third, *safety* states whether failures cause catastrophic consequences. Fourth, *integrity* indicates whether the system achieves its requirements without corrupted or unintended output. Fifth, *maintainability* concerns the flexibility of overpassing modifications and changes. Naturally, fault tolerance impacts these five criteria.

The classification of faults can be based on many criteria. Examples are given in [Tix09]: first, faults are classified according to the timing at which a fault hits the system. In this classification, faults are classified into *transient faults* which seldom hit the system and disappear, *intermittent faults* which frequently hit the system and disappear, and *permanent* faults which stay until the system is maintained. Second, the classification can be based on whether a fault hits a state; i.e. changes some values erroneously, or changes the system's definition, implementation, or behavior. Third, faults can be classified by whether they hit parts of a distributed system or the overall system.

From an outside perspective, fault tolerance is classified into two main categories [TVS07]. First, *masking* fault tolerance, which hides the consequences of faults, and enables the system to continue functioning correctly. This is usually achieved using *replication*. In replication, some components of the system are duplicated, and

the same task can be done by many components, such that if one fails, others can substitute it. Such an approach is often expensive. Second, *non-masking* fault tolerance, which overcomes the consequences of faults on the system's behavior in finite time, however, the system's behavior during the recovery may not be fully controllable. Examples of non-masking fault tolerance are self-stabilization [Dij74, Dol00] and region-adherence [BRT14]. Self-stabilization is described in the following section.

1.2 Self-Stabilization

Self-stabilization [Dij74, Dol00] is a non-masking fault tolerance concept. It is particularly useful when the system's initial configuration is arbitrary, or when the system exhibits transient faults or changes at runtime, such as dynamic topology changes. In other words, self-stabilization addresses systems' failures described as being in an *illegitimate configuration*, but not failures due to run-time changes in systems' designs or implementations.

Self-stabilization ensures that a system's desired behavior is eventually obtained by the system itself and then never voluntarily – i.e. by the system's design – violated regardless of the system's initial behavior. This comprises two properties:

1. *Convergence*: if a system is in an illegitimate configuration, then the system guarantees reaching a legitimate configuration in finite time.

2. *Closure*: if a system is in a legitimate configuration, then the system never reaches an illegitimate configuration by itself.

In centralized systems, self-stabilization can be simply viewed as a reset of the values of the system's variables, to modify the (global) configuration. This can be achieved for some systems by adding e.g. an `if` statement in the code, that checks whether the (global) configuration is legitimate. However, self-stabilization is of a significant impact in distributed systems, where each system component does not have full knowledge about the global configuration of the system. Thus, the components' reactions – with their local knowledge – have to direct the behavior towards satisfying the desired properties of the system [Dij74].

Self-stabilization has tackled many problems in distributed computing. Examples are mutual exclusion [Dij74], spanning tree construction [CYH91], leader election [DIM91], graph coloring [Tur14], token circulation [BGW89], phase clock synchronization (unison) [CFG92], and many others.

There are different models of distributed systems, over which self-stabilization is studied. Three of them are mentioned in the following:

- The first and typical model in this area is the *shared memory* model [Dij74]. This model regards a distributed system as a connected graph of processes linked with edges. Each process has *read and write registers*, and can read (write) the values stored in the read (write) registers of its neighbors. The read and write actions are composite atomic in this model; i.e. if a write action is executed in a step based on some values, then these values are read in the same step. In practice, this model may require some extra setting to guarantee the composite atomicity.

A generalized model of this model has appeared in [DIM93], where the read and write actions are atomic. The shared memory model is formalized in Chapter 2, and is used in Chapters 4, 5, and 7.

- The *message-passing* model [KP90] is a more realistic and popular model. In this model, each component may send (receive) a message to (from) one of its neighbors. The send and receive actions are atomic.
- The *write-all-with-collision* (WAC) model [KA03b] fits wireless networks. In this model, each component may send a message that is received by all its neighbors in one atomic step. However, if two neighbors of a component send messages simultaneously, the component does not react to any of the received messages. The latter case models message collision in wireless networks. Relevant work to this model is given in Chapter 6.

The order of executing actions is determined by the so-called *scheduler* or *daemon*. For example, the *centralized scheduler* assumes that one process executes an action in each step. The *distributed scheduler* assumes that at least one process executes an action in each step. The *synchronous scheduler* assumes that in each step, all processes – intending to execute actions – execute the actions in the same step. The relevant schedulers to this work are formalized in Chapter 2.

Self-stabilization is dependent on the underlying model and scheduler. For example, a self-stabilizing algorithm under the shared memory model with composite atomicity of the read and write actions may not be self-stabilizing if the read and write actions are atomic. Many transformational approaches between models are given in [Dol00]. In addition, a self-stabilizing algorithm under the synchronous scheduler may not be self-stabilizing under the asynchronous scheduler. A *synchronizer* preserving self-stabilization can be a transformational approach for this case [BPV04].

A self-stabilizing system wrt. a property can be based on multiple layered self-stabilizing algorithms, where each algorithm stabilizes wrt. its own property. The algorithms are layered in a scheme, such that the self-stabilization of an algorithm at some layer is not impacted by the algorithms in the upper layers. The stabilization of the whole system, then, starts from the lowest layer, and goes upwards. This scheme is called a *fair composition* of self-stabilizing algorithms [DIM93, Dol00]. For example, a fair composition can be constructed of a spanning tree algorithm (lower layer) and a wave algorithm designed for trees (upper layer). In this case, the wave algorithm may not function correctly until the tree is constructed by the spanning tree algorithm.

The major criteria that define the efficiency of any system are time and space requirements; a system is considered to be more efficient than another system in the same environment if it requires less time and space to achieve its goals. For self-stabilizing distributed systems, besides the space requirement (e.g. [BDPV99c, BDPV99d, BPV04, BPV08]), efficiency is usually defined by two criteria. The first one is the worst-case of convergence time among all possible executions to reach a legitimate configuration, starting from any configuration. The time complexity can be measured using many metrics. A metric can be number of *rounds*, where a round is defined by a minimum subexecution, in which each process executes an action. Convergence time complexity can also be measured by the number of synchronous or asynchronous

steps, depending on the scheduler and the application of the algorithm. Examples are [KK13, DG13]. In [KK13], the authors confirm the optimal convergence time for any synchronous spanning tree algorithm. In [DG13], the optimal convergence time for any synchronous mutual exclusion algorithm is shown. Second, efficiency is related to the generality of the topologies or schedulers, over which the algorithm is applied. Examples are [BPV08, LMV14]. In [BPV08], the authors show the minimal space required to achieve a self-stabilizing phase-clock synchronization under several topologies using the shared memory model. In [LMV14], the authors show a time-efficient self-stabilizing wave algorithm for non-oriented trees (trees with directed edges but not necessarily directed paths) using the message passing model.

The concept of self-stabilization is also extended to many other concepts. Two of them are mentioned here. The first one is *snap-stabilization* [BDPV99d], in which a specification over the system's behavior is defined, and any behavior of the system should adhere to this specification, starting from any configuration. The specification could be, for example, that if a token is sent from a specific component, the message reaches all components and returns to the initial sender. The second example is *superstabilization* [DH97, Dol00], which has the same requirements as self-stabilization, with an additional characterization to the time required to recover from dynamic topology changes: the superstabilization time is the time required to reach a legitimate configuration after a change only in the topology.

1.3 General Problem Statement

Answering a question in computer science is based on setting an environment, a model, assumptions or restrictions, and applying some techniques to find the solution. The solution can, then, be optimized by e.g. reducing the time and space required to perform the solution's tasks.

As mentioned in the previous section, fault tolerance, in general, impacts the dependability of a system, and dependability is defined over a variety of properties. For example, the availability measures the probability at which a system provides "useful output." Useful output, in this sense, requires that a system achieves its aims correctly, on time, without breaking safety critical requirements. Intuitively, such properties are usually correlated to the time required to accomplish specific tasks.

It is well known that a system's failure due to a transient fault may stop the system to provide its service for some time. Providing service is, however, not limited to safety properties; a system can be safe, but idle forever. For example, a traffic light system might be in a configuration, where it has "Red" light in all directions. This configuration is safe; however, it does not allow the traffic to move. Properties defining a system's progress – to some extent – belong to the class of *liveness properties* [AS87]. A liveness property guarantees that some condition is satisfied infinitely often. However, there is no restriction on the waiting time until the condition is satisfied. In the traffic light example, this condition implies that one direction has "Green" light.

The classical self-stabilization pertains properties that are defined over configurations. The convergence and closure aspects guarantee reaching an execution suffix, in which a property is satisfied by all configurations. Such properties are, in practice, *safety*

properties. From this point, the stabilization of a system or the recovery from transient faults is fixed to safety properties defined over configurations. Moreover, the time efficiency issue in classical self-stabilization is usually a matter of reducing the convergence time to achieve safety. Note that other concepts of self-stabilization concern specifications over the systems' behaviors, like snap-stabilization [BDPV99d]. These concepts, however, do not necessarily follow a specific pattern, and they study particular cases.

In self-stabilization, there is an assumption that faults are transient. In addition, a self-stabilizing system can be practical – e.g. by achieving high availability – under the assumption that faults happen infrequently. With these two assumptions, there is implicitly enough time for each system's behavior to stabilize, and to satisfy the liveness properties, before the next fault hits the system. In other words, these assumptions provide an implicit conclusion that the satisfaction of liveness properties is neither impacted by the length of the convergence time, nor by the time at which a fault hits the system. This fact limits the study of self-stabilization to safety properties defined over configurations, while neglecting the other aspects of a system's performance.

The main question to be investigated in this work is, how can self-stabilization cover properties other than safety properties, like performance: the amount of output the system produces per time unit, efficiency: the amount of output wrt. the amount of input, and quality of service? In other words, how can self-stabilization be generalized to cover other kinds of properties, in which recovery implies not only being safe, but also reaching the desired service? The following is the general problem statement for this work.

Problem 1.1. Provide a generalized model of self-stabilization, that covers properties reflecting efficiency, performance, and quality of service aspects. Study and apply the model over problems in distributed computing. Next, provide an automatic verification approach that simplifies verifying the correctness of systems adhering to this kind of properties.

1.4 General Contribution

The contribution of this work – in light of the general problem statement – is as follows:

- A generalized model of self-stabilization covering performance aspects is introduced. The model involves a general scheme of properties, that are signified as *recurrence properties.* This generalized model of self-stabilization is defined not only for discrete systems, but also for real-time systems to cover examples with real-time aspects (cf. Chapter 6).
- Self-stabilization wrt. recurrence properties is applied to three problems in distributed computing:
    1. *Mutual exclusion*: the generalized concept studies the service time of synchronous mutual exclusion algorithms; i.e. the frequency at which processes are granted a privilege. The study shows trade-off between many aspects: optimizing the convergence time wrt. the safety property of mutual

exclusion, optimizing the frequency to be achieved, and optimizing the convergence time wrt. the achievable frequency. The study provides new synchronous algorithms having new time and space complexities, where some are optimal.

2. *Unique process selection*: this problem is a particular extension of mutual exclusion introduced in this work. The problem differs from mutual exclusion by customizing the fairness requirement: fairness is required to be achieved only under given environments and circumstances. This problem is further extended to *educated unique process selection*, in which a process is granted a privilege only if its state satisfies some local or global requirements. The study involves designing self-stabilizing wave algorithms, based on *propagation of information with feedback (PIF)*.
3. *Time-Division-Multiple-Access (TDMA) slot assignment*: this problem concerns scheduling communication over time, between nodes in wireless sensor networks. The study focuses on reducing and optimizing the length of particular intervals used to avoid collision, namely the *guard time*, in order to increase the communication's throughput. The study tackles real-time networks having tree topology, and provides a formal approach to optimize the guard time for such networks. As a case study, the formal approach is applied to a wireless fire alarm system that conforms to the *European Norm* EN 54-25 [DIN05].

- An automatic verification approach for self-stabilizing systems wrt. recurrence properties is introduced. The approach is based on reducing the problem of verifying properties over infinite executions, to finding counterexamples with a given fixed length. The approach is achieved with the help of model checking. A case study of a mutual exclusion algorithm is provided.

1.5 Outline

The remainder of this thesis is structured as follows. Chapter 2 provides preliminaries, definitions, and general notation. Next, Chapter 3 introduces recurrence properties and the generalized model of self-stabilization. Then, Chapter 4 applies the generalized concept to the mutual exclusion problem. After that, Chapter 5 introduces the educated and unique process selection problems, and applies the generalized concept to solve it. Next, Chapter 6 considers the TDMA slot assignment problem under the generalized concept. After that, Chapter 7 presents the automatic verification approach. Finally, Chapter 8 states the conclusion and future prospects.

2 Preliminaries and Notation

This chapter presents preliminary information, formalism, and models of the topologies and systems considered in the following chapters. The chapter is structured as follows. Section 2.1 presents some basics of number systems. Section 2.2 formalizes graphs, trees, and their properties. Section 2.3 presents a model of discrete systems. Finally, Section 2.4 presents a model of real-time systems.

2.1 Basics

This section defines some notation in regard to number systems and some basic mathematical functions. First, the number systems are given in the following table, which in particular specifies notation for positive numbers including and excluding 0.

Number System	Symbol	Positive, Excluding 0	Positive, Including 0
Real Numbers	$\mathbb{R}$	$\mathbb{R}^+$	$\mathbb{R}_0^+$
Rational Numbers	$\mathbb{Q}$	$\mathbb{Q}^+$	$\mathbb{Q}_0^+$
Integers	$\mathbb{Z}$	$\mathbb{Z}^+$	$\mathbb{Z}_0^+$
Natural Numbers	$\mathbb{N}$	$\mathbb{N}$	$\mathbb{N}_0$

Next, notation of some basic mathematical functions over number systems is defined in the following table. Let *Num* be a number system.

Function Description (if applicable)	Notation
Minimum of *Num*	$\mathsf{min}(Num)$
Maximum of *Num*	$\mathsf{max}(Num)$
Infimum of *Num*	$\mathsf{inf}(Num)$
Supremum of *Num*	$\mathsf{sup}(Num)$
Average of *Num*	$\mathsf{avg}(Num)$

Other than the given basics, other expressions are defined. First, the word "iff" is used to express "if and only if". Second, the symbol "$\mathcal{O}$" denotes the big O notation that is used to describe complexity measures. Next, for each $a, b \in \mathbb{N}_0$, the expression "$a \bmod b$" denotes "a modulo b". Next, let $\star \in \{>, < \,.\, \geq, \leq\}$. The expression $a, b \star c$ denotes $a \star c \wedge b \star c$. Finally, for any quantified statement with $\forall$ or $\exists$, the symbol "$\bullet$" is used to separate the quantifiers from the statement's body, and subsequent quantifiers in one statement are separated with commas. The following statement is an example.

$$\forall\, x, \exists\, y \bullet y \geq x$$

2.2 Topology

This section defines general graphs and trees as underlying topologies for distributed systems. Note that in each following chapter, the topology is specified in the beginning of the chapter. In addition, for all topologies in this work, it is assumed that the vertices have always unique id's.

2.2.1 Vertex

A **vertex** v is a tuple $(\mathsf{id}, \mathit{Vars})$, where $\mathsf{id} \in \{0, \ldots, \mathsf{n}-1\} \subset \mathbb{N}_0$, and *Vars* is a set of variables. A vertex v of id i is denoted by v_i. A variable *var* that belongs to a vertex v_i, is denoted by var_i or var_{v_i}. The *domain* of a variable *var*, denoted by $\mathsf{Dom}(var)$, is the set of all possible values of *var*.

The variables are used to specify the states of the vertices, and therefore, the global configuration of the system.

2.2.2 Graph

A **graph** is a tuple $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a non-empty set of *vertices*, $\mathcal{E} \subseteq \{\{v, v'\} : v, v' \in \mathcal{V} \wedge v \neq v'\}$ is a set of *edges*, and each vertex in $\mathcal{V}$ has a unique id. Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$:

- For each $\{v, v'\} \in \mathcal{E}$, v and v' are said to be *neighbors*. The set of neighbors of a vertex v is denoted by $\mathcal{N}_v$. The symbol $\mathcal{N}_v^*$ is used to denote $\mathcal{N}_v \cup \{v\}$.
- A *path* in $\mathcal{G}$ is a finite sequence of vertices $v_0, \ldots, v_{u-1}$, such that there exists $\{v_i, v_{i+1}\} \in \mathcal{E}$ for each $0 \leq i < u-1$. For each path $v_0, \ldots, v_{u-1}$:
  - The vertex v_0 is said to be *linked* to the vertex v_{u-1} by this path.
  - The number of edges in the path – which equals $u-1$ – is said to be the *length* of the path, denoted by $\mathsf{len}(v_0, \ldots, v_{u-1})$.
- The *degree* of a vertex v, denoted by $\mathsf{degree}(v)$, is the number of edges that are adjacent to v. The *distance* between any two vertices $v, v' \in \mathcal{V}$, denoted by $\mathsf{dist}(\mathcal{G}, v, v')$, is the length of the shortest path in $\mathcal{G}$ that links them. The *diameter* of a graph $\mathcal{G}$, denoted by $\mathsf{diam}(\mathcal{G})$, is the maximum distance between any two vertices. The *size* of $\mathcal{G}$ is $|\mathcal{V}|$, denoted by n (short version of $\mathsf{n}(\mathcal{G})$).

Directed Connected Graph

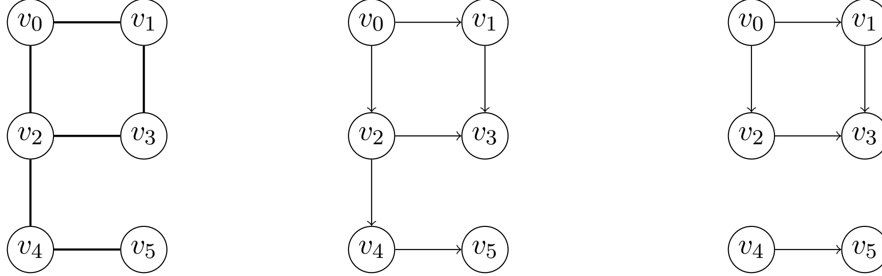
A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is said to be **directed** iff $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ holds, and for each path $v_0, \ldots, v_{u-1}$, for each $0 \leq i < u-1$, $(v_i, v_{i+1}) \in \mathcal{E}$. $\mathcal{G}$ is said to be **connected** iff:

- if $\mathcal{G}$ is undirected, for each two vertices $v, v' \in \mathcal{V}$, there exists at least one path that links them.
- If $\mathcal{G}$ is directed, the undirected version of $\mathcal{G}$ is connected.[1]

[1] Note that this meets the notion of *weakly connected* directed graph.

Assumption. This work concerns distributed systems and related properties. For that, the trivial cases where a graph has a unique node are w.l.o.g. neglected. It is assumed that for each graph $\mathcal{G}$, $\mathsf{n} \geq 2$ holds, which entails that if $\mathcal{G}$ is connected, then $\mathsf{diam}(\mathcal{G}) \geq 1$.

Figure 2.1 presents three graphs as examples. Figure 2.1a shows an undirected connected graph, where both sequences v_5, v_4, v_2, v_0 and v_0, v_2, v_4, v_5 are paths of length 3. The neighbors of v_0 ($\mathcal{N}_{v_0}$) are v_1 and v_2. The diameter equals 4. The graph in Figure 2.1b is directed, and the sequence v_5, v_4, v_2, v_0 is not a path. The graph in Figure 2.1c is not connected since there is no path from any of v_4, v_5 to the others.



(a) Undirected and connected (b) Directed and connected (c) Directed and not connected

Figure 2.1: Examples of graphs of size $\mathsf{n} = 6$

2.2.3 Tree

A **tree** is a directed connected graph $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ with a unique vertex v such that, for each $v' \in \mathcal{V}$, there is a unique path from v to v' in $\mathcal{T}$; v is called *root* of $\mathcal{T}$, denoted by root. Given a tree $\mathcal{T} = (\mathcal{V}, \mathcal{E})$:

- Let $v_0 \in \mathcal{V}$ be the root. For each path $v_0, \ldots, v_{u-1}$, for each $0 \leq i < u-1$, v_i is said to be a *parent* of v_{i+1}, denoted by $\mathsf{parent}(v_{i+1})$, and v_{i+1} is said to be a *child* of v_i. The set of children of v_i is denoted by $\mathsf{Ch}(v_i)$. Each vertex with no children is called a *leaf*, and each vertex that is neither a root nor a leaf is called an *inner vertex.* The set of leaves (resp. inner vertices) in $\mathcal{V}$ is denoted by $\mathsf{Leaves}(\mathcal{T})$ (resp. $\mathsf{Inner}(\mathcal{T})$) – shortly, Leaves (resp. Inner).

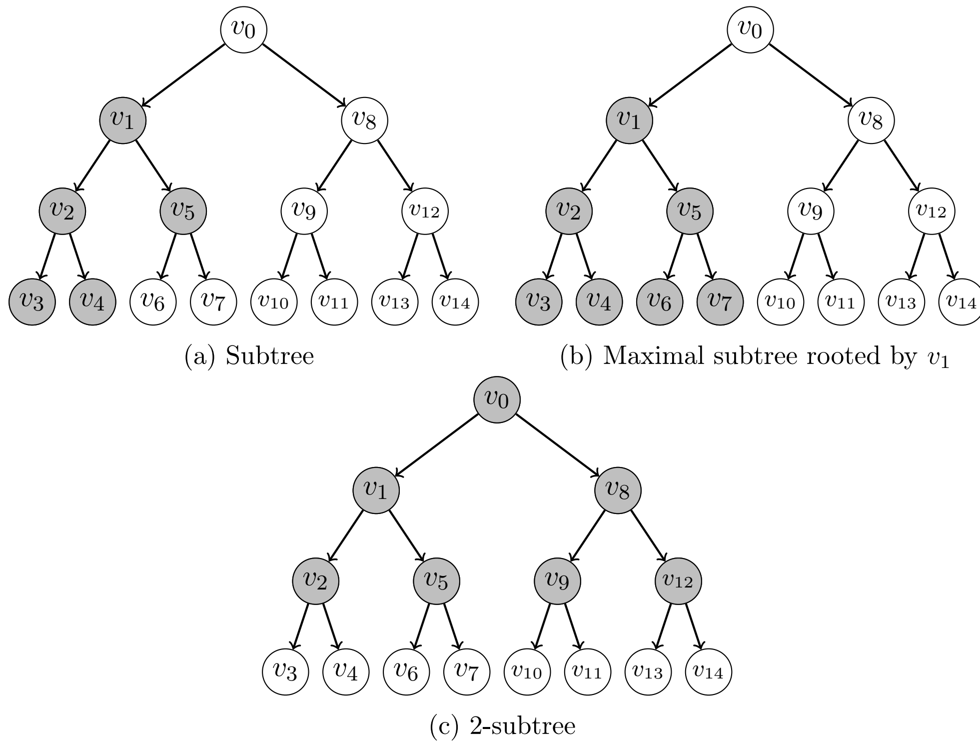
- For each vertex $v \in \mathcal{V}$, the *depth* of v, denoted by $\mathsf{depth}(v)$, is the length of the path that links the root with v. The depth of $\mathcal{T}$, denoted by $\mathsf{depth}(\mathcal{T})$, is the maximum depth observed for any vertex $v \in \mathcal{V}$. $\mathcal{T}$ is said to have a *branching factor* of $f \in \mathbb{N}$ iff $\forall\, v \in \mathcal{V} \setminus \mathsf{Leaves}(\mathcal{T}) \bullet |\mathsf{Ch}(v)| = f$.

- A *subtree* of $\mathcal{V}$, rooted by a vertex v, is a tree $\mathcal{T}' = (\mathcal{V}', \mathcal{E}')$, where $v \in \mathcal{V}' \subseteq \mathcal{V}$, $\mathcal{E}' \subseteq \mathcal{E}$, and v is the root. The subtree $\mathcal{T}'$ is said to be *maximal* iff there is no larger subtree rooted by v. A *d-subtree* of $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ is a subtree $\mathcal{T}' = (\mathcal{V}', \mathcal{E}')$ of $\mathcal{T}$, such that

  1. $\mathcal{T}'$ is rooted by the root,
  2. $\mathsf{depth}(\mathcal{T}') = d$, and
  3. there is no larger subtree of $\mathcal{T}$ having depth d.

Following the assumption that $\mathsf{diam}(\mathcal{G}) \geq 1$ holds for each connected graph $\mathcal{G}$, for each tree $\mathcal{T}$, $\mathsf{depth}(\mathcal{T}) \geq 1$ also holds.

Figure 2.2 illustrates trees of size $\mathsf{n} = 15$, depth 3, branching factor 2, and rooted by v_0. The gray vertices denote subtrees: in Figure 2.2a, the subtree is not maximal, because it is not the largest subtree rooted by v_1; the maximal subtree rooted by v_1 is given in Figure 2.2b. Figure 2.2c illustrates a 2-subtree: it has a depth 2, and there is no larger subtree rooted by the root and having depth 2.



(a) Subtree

(b) Maximal subtree rooted by v_1

(c) 2-subtree

Figure 2.2: Examples of trees of size $\mathsf{n} = 15$ and depth 3

2.3 Discrete Systems

To formalize discrete systems over a graph, the shared memory model of Dijkstra [Dij74] is used, with composite atomicity of the read and write actions. The model is formalized in detail in the upcoming sub-sections. Briefly, the model reflects distributed algorithms executed by processes forming a graph, where the registers of each process are visible to its neighbors. For simplicity, the read and write registers are abstracted by variables. Each process runs a sub-algorithm of a given distributed algorithm. A sub-algorithm is a set of so-called guarded commands that are enabled depending on the states of the process and its neighbors. If a process has an enabled guarded command, the process executes an action that modifies the variables of the process. The order and concurrency of executing actions is abstracted by the so-called scheduler.

2.3.1 Configurations and Executions

A **state** s of a vertex v_i is a vector of the values of the vertex's variables and id:

$$s \in \mathsf{Dom}(var_0) \times \cdots \times \mathsf{Dom}(var_{\mathsf{n}-1}) \times \{0, \ldots, \mathsf{n}-1\}.$$

A **configuration** of a graph $\mathcal{G}$, denoted by γ, is a vector of the states of all vertices in $\mathcal{G}$. The set of all possible configurations is called the *configuration space*, denoted by Γ (short version of $\Gamma(\mathcal{G})$). A configuration γ is said to satisfy a condition con, denoted by $\gamma \models \mathsf{con}$, iff con evaluates to *true* under γ.

An **execution** $\varXi$ over a graph $\mathcal{G}$ is a sequence of configurations of $\mathcal{G}$, which can be *finite* $\gamma_0, \ldots, \gamma_{u-1}$ or *infinite*. Let $\varXi : \gamma_0, \gamma_1, \ldots$ be an execution, and let $i, j \in \mathbb{N}_0$.

- A *step of* $\varXi$ is a pair (γ_i, γ_{i+1}).
- A *subexecution* of $\varXi$ is a finite subsequence $\gamma_i, \ldots, \gamma_j$ of $\varXi$, where $j \geq i$.
- An *execution prefix* of $\varXi$ is a finite subexecution $\gamma_0, \ldots, \gamma_j$.
- An *execution suffix* of $\varXi$ is a subexecution $\gamma_i, \gamma_{i+1}, \ldots$.

Let $\varXi : \gamma_0, \ldots, \gamma_{u-1}$ be a finite execution.

- A *strict subexecution* of $\varXi$ is a finite subsequence $\gamma_i, \ldots, \gamma_j$ of $\varXi$, such that $i > 0$ or $j < u - 1$.
- The *length* of $\varXi$, denoted by $\mathsf{length}(\varXi)$, is u; i.e., the number of configurations in $\varXi$.[1]

A **system** $\varOmega$ (discrete system) is a – possibly infinite – set of executions, such that for each subexecution $\varXi'$ of an execution $\varXi \in \varOmega$, $\varXi'$ is an execution prefix of an execution $\varXi'' \in \varOmega$. In other words, it is assumed that any configuration is initial configuration of an execution in the system. This assumption is required to study the properties of self-stabilization, where a fault may change the configuration arbitrarily, and the system should stabilize starting from the new configuration as if it is initial.

2.3.2 Distributed Algorithms

A **distributed algorithm** is a set of *sub-algorithms* $\{A_0, \ldots, A_{u-1}\}$. Each sub-algorithm A_i is a set of *guarded commands* – shortly *commands* – $\{\mathsf{gc}_0, \ldots, \mathsf{gc}_{y-1}\}$, such that $\mathsf{gc} : guard \longrightarrow action$, where:

- gc is called a *label*. The label is any unique expression.
- *guard* is a Boolean expression over variables.
- *action* is a set of assignment functions.

[1] In contrast, the length of a path pa is the number of edges in the path, denoted by $\mathsf{len}(pa)$.

A **process** p is an extended vertex $(\mathsf{id}_p, \mathit{Vars}_p, A_{subp})$, where A_{subp} is a sub-algorithm. Let $\mathcal{G} = (\mathcal{P}, \mathcal{E})$ be a graph, where $\mathcal{P}$ is a set of *processes*. Each process p_i runs a sub-algorithm A_i, such that for each command $\mathsf{gc} : \mathit{guard} \longrightarrow \mathit{action}$ of A_i, *guard* is a Boolean expression over the variables of all $p_j \in \mathcal{N}^*_{p_i}$, and *action* can modify only the values of the variables belonging to p_i. A command $\mathsf{gc} : \mathit{guard} \longrightarrow \mathit{action}$ is said to be *enabled* in a configuration γ iff *guard* evaluates to *true* in γ. A process p_i is said to be *enabled* in γ iff it has an enabled command in γ.

An execution of a distributed algorithm over a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is an execution $\gamma_0, \gamma_1, \ldots$ where in each step (γ_i, γ_{i+1}) for $i \geq 0$, a non-empty subset of enabled processes in $\mathcal{P}$ execute the actions of their enabled commands, where each process of them may execute actions of multiple enabled commands.[1]

2.3.3 Schedulers

A *scheduler* is simply a restriction on the set of executions. It can also be described as a daemon that chooses which actions to be executed in each step. In this work, three types of schedulers are formalized by describing the corresponding executions.

Let $\mathcal{G}$ be a graph, let A be a distributed algorithm, and let $\Xi : \gamma_0, \gamma_1, \ldots$ be an execution of A over $\mathcal{G}$.

- The execution Ξ is said to be **asynchronous** iff for each step (γ_i, γ_{i+1}) where $i \geq 0$, there exists at least one enabled process p in γ_i that executes the actions of a subset of enabled commands of p in (γ_i, γ_{i+1}). Asynchronous executions refer to the so-called *distributed scheduler*.

- The execution Ξ is said to be **synchronous** iff for each step (γ_i, γ_{i+1}) where $i \geq 0$, each enabled process executes actions in (γ_i, γ_{i+1}), given that at least one process executes its actions. This refers to the *synchronous scheduler*.

- The execution Ξ is said to be **fair** iff for each process p, if p is enabled in a configuration γ_i, then p executes actions in a step (γ_j, γ_{j+1}) for $j \geq i$. This refers to the *fair scheduler*.

2.3.4 Self-Stabilization

Self-stabilization wrt. some condition con comprises the two properties *convergence* and *closure*. The former property denotes that a configuration satisfying con is reached regardless of the initial configuration, and the latter denotes that a configuration violating con is never reached starting from a configuration satisfying con. In the following, self-stabilization is formalized in terms of executions.

A system Ω is said to be **self-stabilizing** wrt. a condition con iff for each execution $\gamma_0, \gamma_1, \ldots$, there exists finally a configuration γ_j, for $j \geq 0$, such that:

- Each configuration of the execution suffix $\gamma_j, \gamma_{j+1}, \ldots$ satisfies con.

- Each configuration of the execution prefix $\gamma_0, \ldots, \gamma_{j-1}$ (if existing) violates con.

[1] This corresponds to the notion of *concurrent executions*, which contrasts *serial executions* where each process may execute the actions of exactly one enabled command in each step.

The **convergence time wrt.** con – or the con**-convergence time** – of an execution $\gamma_0, \gamma_1, \ldots$ is the minimum $i \in \mathbb{N}_0$ such that each of $\gamma_i, \gamma_{i+1}, \ldots$ satisfies con. The con-*convergence time* of a system Ω is the minimum $j \in \mathbb{N}_0$, such that for each execution $\Xi : \gamma_0, \gamma_1, \ldots$ in Ω, each of the configurations $\gamma_j, \gamma_{j+1}, \ldots$ satisfies con.

For example, let $\Xi : \gamma_0, \gamma_1, .., \gamma_5, \gamma_6, ...$ be an infinite execution, where the configurations $\gamma_5, \gamma_6, ...$ satisfy a condition con, and the configurations $\gamma_0, ..., \gamma_4$ do not satisfy con. The con-convergence time of Ξ is 5. If, for example, γ_3 also satisfies con, then Ξ does not converge wrt. con by the given classical definition, because γ_4 does not satisfy it.

2.4 Real-Time Systems

A real-time systems is defined by a set of evolutions – confronting executions in discrete systems – over time. The following formalism of real-time systems is inspired from [OD08]. Self-stabilization is also defined for real-time systems in terms of evolutions over time.

By Time, the infinite set $\mathbb{R}_0^+ = [0, \infty)$ is denoted. Let $\mathcal{G}$ be a graph, and let Γ denote the set of all configurations of $\mathcal{G}$. An **interpretation**[1] $\mathcal{I}$ of $\mathcal{G}$ at a point in time $t \in \mathsf{Time}$ is the valuation of all the variables and ids observed in $\mathcal{G}$ at t; i.e.

$$\mathcal{I} : \mathsf{Time} \longrightarrow \Gamma$$

An interpretation over a time interval $[t_1, t_2] \subset \mathsf{Time}$, denoted by $\mathcal{I}([t_1, t_2])$, is the set of interpretations at all points in $[t_1, t_2]$. An **evolution** over $\mathcal{G}$ is an *interpretation* of $\mathcal{G}$ over time.

In the definition of discrete systems, it is assumed that each configuration is initial, in order to consider self-stabilization. A similar assumption is included for real-time systems as well. A **real-time System** RTS over $\mathcal{G}$ is a set of evolutions over a graph $\mathcal{G}$, under the assumption that for each evolution $\mathcal{I} \in \mathsf{RTS}$ and each time point $t \in \mathsf{Time}$, there exists an evolution $\mathcal{I}' \in \mathsf{RTS}$ such that $\mathcal{I}(t) = \mathcal{I}'(0)$.

A real-time system RTS is said to be **self-stabilizing** wrt. a condition con iff for each evolution $\mathcal{I} \in \mathsf{RTS}$, there exists a time point $t \in \mathsf{Time}$, such that:

- For each $t' \geq t$, the condition con is satisfied by $\mathcal{I}(t')$.
- For each $t'' < t$ (if existing), the condition con is violated by $\mathcal{I}(t'')$.

The convergence time of an evolution $\mathcal{I}$ wrt. a condition con is the minimum $t \in \mathsf{Time}$ such that $\mathcal{I}(t)$ satisfies con. The convergence time of a real-time system Ω wrt. con is the minimum t such that for each evolution $\mathcal{I} \in \Omega$, the condition con is satisfied by $\mathcal{I}(t)$.

Figure 2.3 illustrates an example of an evolution $\mathcal{I}$ over a topology $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where each vertex v_i has a variable x_i. It states that for all vertices v_i, the value of x_i equals 0 in the interval $[0, t)$, and equals 1 starting from t. Let con be a condition defined as follows:

$$\mathsf{con} := \forall\, v \in \mathcal{V} \bullet v.x = 1$$

By the definition of self-stabilization, the evolution $\mathcal{I}$ is self-stabilizing wrt. con in t time.

[1]"Interpretation" meets the notion of configuration in discrete systems.

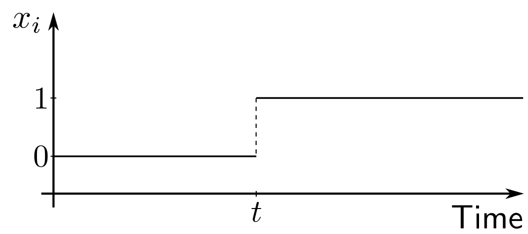


Figure 2.3: An evolution $\mathcal{I}$

The presented formalism in this chapter provides a basis for modelling the discrete and real-time systems and algorithms in the following chapters, in addition to their correctness proofs.

3 Recurrence in Self-Stabilization

This chapter introduces a new property to be considered in the scope of self-stabilization, which is the core property concerned in this thesis. The property comprises a performance measure, defined over executions in contrast to the classical properties defined over configurations in self-stabilization. Some of this chapter's content has appeared in [3, 7] of the author's publications.

This chapter is structured as follows. Section 3.1 motivates the property. Next, Section 3.2 formalizes the property and extends the definition of self-stabilization for discrete systems. Lastly, Section 3.3 formalizes the property for real-time systems.

3.1 Beyond Safety in Self-Stabilization

System requirements or properties are usually categorized into safety and liveness properties. On one hand, satisfying safety properties guarantees that the system does not violate critical requirements. On the other hand, satisfying liveness properties ensures that the system progresses, provides the aimed output, and executes useful actions.

Whether a system is efficient or not can be specified by many criteria. For examples, this can be measured by how fast the system responds, how high the throughput is, how available the system is, the amount of energy conserved by the system, and others. As a comprehensive view, the efficiency can be defined by the degree of useful output provided by the system, given some amount of time and energy. A useful output of a system is usually the result of actions, that are executed when some certain liveness properties are satisfied.

Liveness properties cannot be violated in finite time as safety properties. In detail, satisfying liveness guarantees that some actions are executed, no matter when. However, satisfying safety requires satisfying conditions within deadlines on time or number of steps. This has two implicit impacts in self-stabilization:

1. The convergence time of a system is actually the time required to reach a behavior in which the safety property always holds, no matter when the system satisfies the conditions related to liveness properties. This fact limits the convergence time optimization to safety-related properties.

2. A liveness property can be satisfied by two systems; however, not necessarily providing the same performance, even if the performance measurement is based on the conditions related to the liveness property. This fact limits the stabilization behavior to satisfy safety and liveness, while neglecting the achieved performance.

The performance in this work is defined by how frequent some condition is satisfied in an execution, which provides the opportunity to measure how frequent an action is executed. Such a definition can be viewed as an abstraction to the typical notion

of throughput. The interesting point to consider for a self-stabilizing system is the (convergence) time required to reach a minimum quality of service, starting from any initial configuration. This provides a generalized view to self-stabilization, and highlights a possible trade-off: in the design of self-stabilizing systems, minimizing the convergence time wrt. safety may increase the convergence time wrt. the desired quality of service, and vice versa.

Figure 3.1 illustrates this issue. It presents two executions Ξ_1 and Ξ_2 (represented by right arrows), where two conditions con and con' are satisfied by some configurations (represented by the gray area and vertical bars) in the executions. The condition con reflects a classical condition in self-stabilization: once con is satisfied by a configuration, any following configuration satisfies con. The condition con' is related to the property reflecting the performance in this work. There are two phenomena to be concerned. First, the convergence time wrt. con is in Ξ_1 smaller than in Ξ_2, while the time required to start satisfying con' in Ξ_2 – from the beginning – is smaller than in Ξ_1. Second, the frequency of satisfying con' at some point in Ξ_2 is greater than in Ξ_1. There is a clear trade-off between the two executions: Ξ_1 is more efficient wrt. con, while execution Ξ_2 is more efficient wrt. con'. In the following, two motivational examples map the conditions con, con', and the scenario in Figure 3.1 to well known problems.

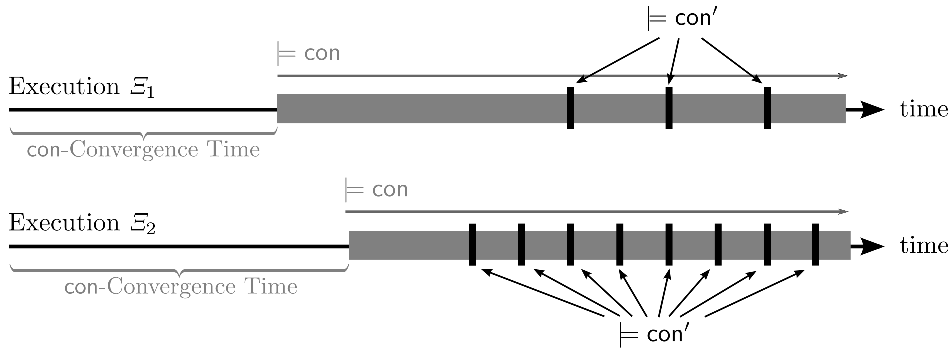


Figure 3.1: Convergence wrt. many properties

The first example concerns the problem of self-stabilization wrt. mutual exclusion [Dij74]. The safety property for mutual exclusion is that at most one process is granted a privilege in each configuration (con). The performance is measured by how often an arbitrary process is granted a privilege in any execution (con' denotes that one process is granted privilege.) Such a performance measure is usually referred to as *service time* [Joh02, Joh04]. Considering mutual exclusion algorithms in synchronous environments: the algorithms of Dijkstra [Dij74] converge wrt. the safety property in n steps – n is the number of processes. If the algorithms are applied to ring topologies, then in each step after convergence, one process is granted a privilege. In contrast, given a graph $\mathcal{G}$, the algorithm in [DG13] converges in $\lceil \mathsf{diam}(\mathcal{G})/2 \rceil$ steps, where $\mathsf{diam}(\mathcal{G})$ is the diameter of $\mathcal{G}$. This convergence time complexity improves the one of Dijkstra's algorithms. However, the algorithm in [DG13] achieves less performance than [Dij74] for ring topologies. Mutual exclusion is considered in detail in Chapter 4, and a related problem to mutual exclusion is presented in Chapter 5.

The second example concerns the slot assignment problem in Time Division Multiple

Access (TDMA) protocols [KA04, HT04, AK05]. TDMA slot assignment is a known problem in distributed computing. It concerns scheduling communication between nodes of a network when there is a limited communication bandwidth. The schedule is achieved by dividing time into slots, in each, a number of particular nodes are assigned to send or listen to messages. The safety property in such systems is that at each point in time, at most one node within some distance in the graph is allowed to send a message (con). The liveness property is that each node frequently has a chance to send its messages. For this example, the system converges when all nodes are assigned to slots. Optimizing the convergence time implies having all nodes assigned as quickly as possible. However, performance could be measured by how often a message of a particular sort is being sent without collision (con' denotes that one node is sending a message of a particular sort.) This performance is usually impacted by the slot assignment order (cf. Chapter 6), and an efficient message delivery in this sense may not be achieved together with fast slot assignment. TDMA slot assignment is considered in detail in Chapter 6.

The following section summarizes related work.

3.1.1 Related Work

The general notion of performance or efficiency in self-stabilization is usually measured according to the convergence time, the space requirement, and the generality of the underlying topology and scheduler. Tremendous work focuses on minimizing the convergence time of self-stabilizing algorithms, e.g. [KK13, DG13]. Other works consider minimizing the space requirement, as in [BDPV99c, BDPV99d, BPV04, BPV08]. Others exploit variant environments, e.g. underlying topologies or schedulers, to enhance the performance [BPV08, CPVD01, LMV14].

There are other examples considering other notions of performance, such as [FBT13, NKM06, DTW06, GGHP96]. The work of [FBT13] presents a metric for measuring the expected mean value of the convergence time. This value reflects the average convergence time, and it can be computed by probabilistic model checking. In [NKM06], the occurrence of transient faults during convergence and their effect on the convergence time is considered. The work of [DTW06] defines and applies fault tolerance metrics, such as reliability and availability, to evaluate self-stabilizing systems, also under the assumption of ongoing faults. The work of [GGHP96] considers the global consequences of a local failure caused by a transient fault until the system converges. A space-efficient transformational approach that reduces the impact of local failures on the global performance is given in [KT12]. These approaches, and many others, basically address performance aspects of self-stabilizing systems.

3.2 Recurrence in Discrete Systems

This section formalizes a property for discrete systems, and generalizes the definition of self-stabilization to cover this property. The topologies considered in this chapter are *connected graphs*.

3.2.1 Recurrence Property

The core measure, on which the performance depends, is basically applied to finite executions: given a finite execution, this measure is the ratio of configurations that satisfy some condition in the execution. This ratio is signified as the *recurrence* of the condition in the execution.

Definition 3.1 (Recurrence)**.** Let $\mathcal{G}$ be a topology, let con be a condition over the configurations of $\mathcal{G}$, and let Ξ be a finite execution over $\mathcal{G}$. The **recurrence** of con in Ξ, denoted by $\mathsf{Rec}_{\mathsf{con}}(\Xi)$, is the ratio $\Delta \in [0,1] \subset \mathbb{R}$ of the configurations satisfying con in Ξ. ♢

For example, let con be a condition defined over configurations of a topology. As notation, let $\underline{\gamma}$ denote that γ satisfies con. For the following finite execution:

$$\Xi : \underline{\gamma_0}, \underline{\gamma_1}, \gamma_2, \gamma_3, \underline{\gamma_4}, \gamma_5, \underline{\gamma_6}, \gamma_7, \gamma_9, \underline{\gamma_9}, \gamma_{10}, \underline{\gamma_{11}},$$

the recurrence of con in Ξ is the result of a simple division operation

$$\mathsf{Rec}_{\mathsf{con}}(\Xi) = \frac{6}{12} = 0.5.$$

Having the notion of recurrence over finite executions, the performance-related property is defined over possibly infinite executions as follows: given an infinite execution Ξ, the property indicates that each execution prefix of Ξ has at least Δ recurrence of the condition con. The property is denoted by con_Δ.

Definition 3.2 (con_Δ)**.** Let con be a condition over the configurations of a topology, and let $\Delta \in [0,1] \subset \mathbb{R}$. An execution $\Xi : \gamma_0, \gamma_1, \ldots$ over a topology is said to *satisfy* $\mathbf{con}_\Delta$ iff for each $i \geq 0$, the recurrence of con in $\gamma_0, \ldots, \gamma_i$ ($\mathsf{Rec}_{\mathsf{con}}(\gamma_0, \ldots, \gamma_i)$) is greater or equal to Δ. ♢

Corollary 3.1. If an execution Ξ satisfies con_Δ for any $\Delta \geq 0$, it follows that for all $0 \leq \Delta' \leq \Delta$, Ξ satisfies $\mathsf{con}_{\Delta'}$.

3.2.2 con_Δ-Convergence Time

Next, the definition of self-stabilization is generalized to cover the property con_Δ. The main thing to be concerned with is the worst-case convergence time to a configuration γ_{c}, from which any execution $\gamma_{\mathsf{c}}, \gamma_{\mathsf{c}+1}, \ldots$ satisfies con_Δ.

Definition 3.3 (con_Δ-Convergence Time)**.** Given a system Ω over a topology, a condition con over the configurations of the topology, and $\Delta \in [0,1] \subset \mathbb{R}$.

- An execution $\Xi : \gamma_0, \gamma_1, \cdots \in \Omega$ is said to **have a con_Δ-convergence time of c steps** iff c is the minimum number, such that the execution suffix $\gamma_{\mathsf{c}}, \gamma_{\mathsf{c}+1}, \ldots$ of Ξ satisfies con_Δ.
  It is also said that for each $j \geq \mathsf{c}$, the execution Ξ **guarantees** con_Δ-convergence in j steps.
- The system Ω is said to **have a con_Δ-convergence time of c steps** iff c is the maximum con_Δ-convergence time among all executions of Ω. ♢

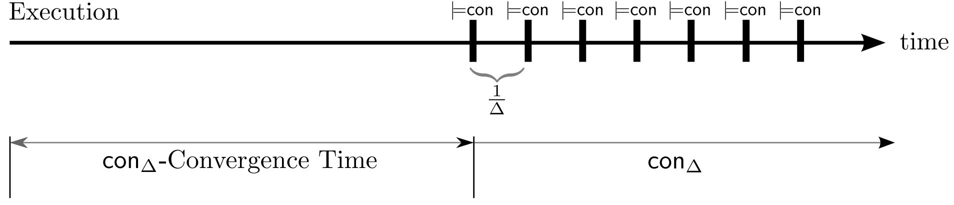


Figure 3.2: con_Δ-Convergence

Figure 3.2 illustrates the intuition of the given definitions by an example of an execution, where the distance between configurations satisfying con is constant.

In Definition 3.3, the two terms *have* and *guarantee* are used to distinguish two slight different meanings. This is done explicitly in the definition to highlight a major difference between the classical self-stabilization wrt. con and self-stabilization wrt. con_Δ. In classical self-stabilization, both *having* and *guaranteeing* a con-convergence time of c steps imply that each configuration of the execution suffix $\gamma_{\mathsf{c}}, \gamma_{\mathsf{c}+1}, \ldots$ satisfies con. However, in Definition 3.3, *having* con_Δ-convergence time of c steps implies that the execution suffix $\gamma_{\mathsf{c}}, \gamma_{\mathsf{c}+1}, \ldots$ satisfies con_Δ, but *guaranteeing* does not necessarily imply it for the exact suffix: *guaranteeing* implies that there exists $0 \leq i \leq \mathsf{c}$ such that $\gamma_i, \gamma_{i+1}, \ldots$ satisfy con_Δ. Note that in a self-stabilizing system wrt. con_Δ in c steps, at least one execution *has* con_Δ-convergence time of c steps (cf. Corollary 3.1).

To illustrate Definition 3.3, consider the three executions given in Table 3.1:

i	Execution i (Ξ_i)	$\mathsf{con}_{0.25}$-Convergence Time for Ξ_i
1	$\Xi_1 : \underline{\gamma_0}, \gamma_1, \gamma_2, \gamma_3, \underline{\gamma_4}, \gamma_5, \gamma_6, \gamma_7, \underline{\gamma_8}, \gamma_9, \gamma_{10}, \gamma_{11}, \underline{\gamma_{12}}, \gamma_{13}, \cdots$	0
2	$\Xi_2 : \gamma_0, \gamma_1, \gamma_2, \gamma_3, \gamma_4, \underline{\gamma_5}, \gamma_6, \gamma_7, \gamma_8, \underline{\gamma_9}, \gamma_{10}, \gamma_{11}, \gamma_{12}, \underline{\gamma_{13}}, \cdots$	5
3	$\Xi_3 : \gamma_0, \gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5, \gamma_6, \underline{\gamma_7}, \underline{\gamma_8}, \underline{\gamma_9}, \underline{\gamma_{10}}, \underline{\gamma_{11}}, \underline{\gamma_{12}}, \underline{\gamma_{13}}, \cdots$	7

Table 3.1: Executions converging to $\mathsf{con}_{0.25}$. $\underline{\gamma}$ denotes that γ satisfies con.

1. Ξ_1: the condition con is satisfied once in every four subsequent configurations, starting from γ_0. By Definitions 3.2 and 3.3, $\mathsf{con}_{0.25}$ is achieved in 0 steps. Indeed, 0.25 is the minimum Δ guaranteed in Ξ_1. If, for example, a transient fault happens after any configuration, say γ_j where $j > 0$, a recurrence of 0.25 is still guaranteed in $\gamma_0, \ldots, \gamma_j$.

2. Ξ_2: the condition con is satisfied in every four subsequent configurations, starting from γ_5. This implies that $\mathsf{con}_{0.25}$ is achieved in five steps.

3. Ξ_3: starting from γ_7, the recurrence of con is 1.0. This case models the classical self-stabilization; i.e. the $\mathsf{con}_{1.0}$-convergence time is 7. Note that by Definition 3.3 and Corollary 3.1, the $\mathsf{con}_{0.25}$-convergence time is also 7 in Ξ_3.

Assuming that the three executions Ξ_1, Ξ_2, Ξ_3 are the only ones for a system Ω, then by Definition 3.3, the con_Δ-convergence time of Ω is 7.

3.2.3 con_Δ-WarmUp Time

In self-stabilization, it is usually assumed that faults do not happen very frequently. In other words, the convergence time is assumed to be small enough such that some service is delivered before another fault hits the system. This, in turn, directs the focus to neglect the delivered service during convergence. In practice, there are environments – like wireless sensor networks – where systems are exposed to high frequency of faults. Naturally, for such environments, it is important that the system provides a reasonable service, even during the convergence of the system.

Such environments are also considered in this work, together with the notion of recurrence. To this end, the notion of con_Δ-*warmup time* is introduced. It denotes the time required by an execution to reach a configuration γ, such that the recurrence of con in the execution prefixes ending in γ and all following configurations is greater or equals Δ. In contrast to the con_Δ-convergence time, the con_Δ-warmup time considers the recurrence of con starting from the beginning of executions.

Definition 3.4 (con_Δ-WarmUp Time)**.** Given a system Ω over a topology, a condition con over the configurations of the topology, and $\Delta \in [0,1] \subset \mathbb{R}$.

- An execution $\Xi : \gamma_0, \gamma_1, \dots \in \Omega$ is said to **have a con_Δ-warmup time of w steps** iff w is the minimum number, such that for each $i \geq \mathsf{w}$, the recurrence of con in the execution prefix $\gamma_0, \dots, \gamma_i$ of Ξ (i.e. $\mathsf{Rec}_{\mathsf{con}}(\gamma_0, \dots, \gamma_i)$) is greater or equals Δ. It is also said that the execution Ξ **guarantees** con_Δ-warmup in i steps.

- The system Ω is said to **have a con_Δ-warmup time of w steps** iff w is the *maximum* con_Δ-warmup time among all executions of Ω. ♢

Figure 3.3 illustrates the warmup notion for an execution Ξ. In the early execution prefixes of Ξ, the recurrence of con does not reach the value Δ. However, at some configuration (say γ_w), the recurrence of con from the initial configuration (γ_0) until γ_w reaches Δ, and then, for any following configuration, the recurrence of con from γ_0 is greater than or equals Δ. This implies that the con_Δ-warmup time is w steps.

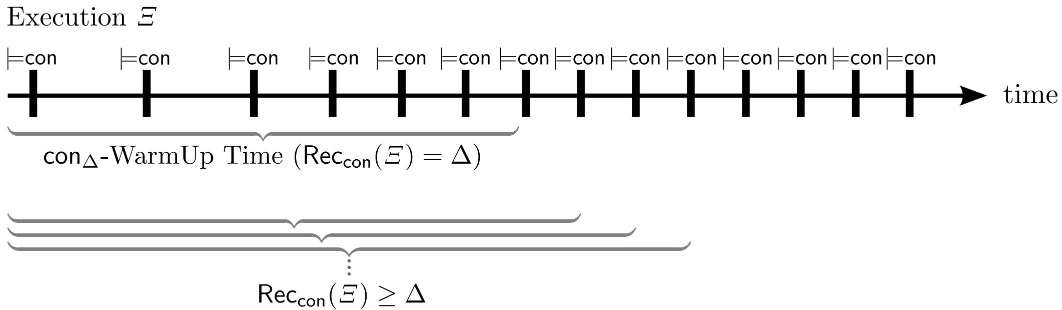


Figure 3.3: con_Δ-WarmUp

3.3 Recurrence in Real-Time Systems

Recurrence is defined for real-time systems similar to discrete systems, except that the ratio is defined over finite time intervals instead of finite executions.

The evolutions of a real-time system are defined over time, which does not exist in the model of discrete systems. For that, the following definition introduces a supplementary function, which specifies whether an evolution satisfies a condition at some point in time.

Definition 3.5. Given an evolution $\mathcal{I}$, a condition con, and a point in time $t \in \mathsf{Time}$, the function **holds($\mathcal{I}, t, \mathsf{con}$)** is defined piecewise as follows:

$$\mathsf{holds}(\mathcal{I}, t, \mathsf{con}) = \begin{cases} 1 & \text{if} \quad \mathcal{I}(t) \models \mathsf{con} \\ 0 & \text{if} \quad \mathcal{I}(t) \not\models \mathsf{con} \end{cases}$$

♢

In order to define the recurrence of con in an evolution $\mathcal{I}$ over a time interval, it is required to measure the length of the sub-intervals in which the con holds; or in other words, in which the function $\mathsf{holds}(\mathcal{I}, \cdot, \mathsf{con})$ is equal to one. However, time is defined by the set $\mathbb{R}_0^+$, and this allows having infinite set of sub-intervals in which $\mathsf{holds}(\mathcal{I}, \cdot, \mathsf{con})$ is equal to one. In practice, the condition con is supposed to hold in a finite number of sub-intervals of an interval. Therefore, it is assumed that for each interval, there is a finite number of sub-intervals, in which $\mathsf{holds}(\mathcal{I}, \cdot, \mathsf{con})$ equals 1. With this assumption, the function $\mathsf{holds}(\mathcal{I}, \cdot, \mathsf{con})$ is integrable.

The following definition formalizes the notions of recurrence and con_Δ for real-time systems.

Definition 3.6. Let con be a condition, and let $\mathcal{G}$ be a topology.

- Given an evolution $\mathcal{I}$ of $\mathcal{G}$ over an interval $[t_1, t_2]$, the **recurrence** of con in $\mathcal{I}([t_1, t_2])$, denoted by $\mathsf{Rec}_{\mathsf{con}}(\mathcal{I}([t_1, t_2]))$ is the ratio of the accumulative time in which con holds in $\mathcal{I}([t_1, t_2])$; i.e.

$$\mathsf{Rec}_{\mathsf{con}}(\mathcal{I}([t_1, t_2]) = \frac{\int_{t_1}^{t_2} \mathsf{holds}(\mathcal{I}, t, \mathsf{con})\ dt}{t_2 - t_1}.$$

- Let $\Delta \in [0, 1] \subset \mathbb{R}$. An evolution $\mathcal{I}$ of $\mathcal{G}$ over a finite interval $[t_1, t_2]$ or an infinite interval $[t_1, \infty)$ is said to *satisfy* con_Δ iff for each $t \geq t_1$ (and $t \leq t_2$ in the former case), the recurrence of con in $\mathcal{I}([t_1, t])$ is greater than or equals Δ. ♢

The following definition models the notion of con_Δ-convergence for real-time systems.

Definition 3.7. Given a real-time system RTS over a topology, a condition con, and $\Delta \in [0, 1] \subset \mathbb{R}$.

- An evolution $\mathcal{I} \in \mathsf{RTS}$ is said to **have a con_Δ-convergence time** $c \in \mathsf{Time}$ iff c is the minimum time point, such that $\mathcal{I}([c, \infty))$ satisfies con_Δ.
  It is also said that for each $j \geq c$, the evolution $\mathcal{I}$ **guarantees** con_Δ-convergence at the time point j.

- RTS is said to have a con_Δ**-convergence time** $c \in \mathsf{Time}$ iff c is the maximum con_Δ-convergence time for all evolutions of RTS. ♢

Similarly, the notion of warmup time is defined for real-time systems.

Definition 3.8. Given a real-time system RTS over a topology, a condition con, and $\Delta \in [0, 1] \subset \mathbb{R}$.

- An evolution $\mathcal{I} \in \mathsf{RTS}$ is said to **have a** con_Δ**-warmup time** $\mathsf{w} \in \mathsf{Time}$ iff w is the minimum time point, such that for each $i \geq w$, the recurrence of con in $\mathcal{I}([0, i])$ (i.e. $\mathsf{Rec}_{\mathsf{con}}(\mathcal{I}([0, i])))$ is greater or equals Δ.
  It is also said that $\mathcal{I}$ **guarantees** con_Δ-warmup at the time point i.
- RTS is said to **have a** con_Δ**-warmup time** $\mathsf{w} \in \mathsf{Time}$ iff w is the *maximum* con_Δ-warmup time among all evolutions of the RTS. ♢

3.4 Remarks

Some remarks regarding recurrence properties are provided.

- The following remarks are directly implied from Definition 3.3 and the definition of self-stabilization:
  - The classical self-stabilization is a special case in which is reflected by setting $\Delta = 1.0$.
  - If the concerned recurrence Δ is greater than 0, the convergence time wrt. con_Δ *guarantees* reaching at least one configuration that satisfies the condition.
  - If the gap between two subsequent configurations satisfying con is constant, say v, as in case Ξ_2 in Table 3.1, the minimum recurrence Δ to be guaranteed after convergence can be computed by a simple division operation: $\frac{1}{v}$.
  - If a condition is not satisfied infinitely often; i.e. if a condition is never satisfied after some configuration, there is no minimum recurrence to be guaranteed. This also holds if the number of configurations between any two subsequent configurations satisfying the condition keep increasing.
- There is a category of self-stabilizing systems, in which the executions of a system are finite, and the system stabilizes at the terminal configurations. Such systems are called silent self-stabilizing systems [DGS96, CD94]. For this category, the recurrence notion may not be useful, since the executions are finite, and the system does not need to execute actions after convergence. However, the case is still covered by Definition 3.3.
- If the system is reactive, i.e. is based on *requesting* and *serving* actions, analyzing the recurrence of *serving* may require to assume that there are always processes that are *requesting*.
- The related work does not offer an explicit proving scheme for checking warmup properties. However, in Chapter 7, automatic verification tools are shown to be useful for this issue.

Recall that in this work, distributed algorithms are defined by guarded commands. To measure the frequency of executing particular commands, it is sufficient to analyze the recurrence of the conditions that guarantee their execution, where the conditions can be defined according to the guards of the commands. This in turn provides the ability to analyze the recurrence of running particular actions in a system. Note that considering the recurrence of the actions is similar to considering the system throughput.

4 Mutual Exclusion

In this chapter, the generalized concept of self-stabilization is applied to design efficient self-stabilizing algorithms wrt. *mutual exclusion*, which is a well known problem in the area of self-stabilization. In the scope of this work, the efficiency reflects having high recurrence of granting a privilege to processes, which implies having a high service time of the algorithms. The major contribution of this chapter is a self-stabilizing mutual exclusion algorithm for connected graphs using the shared memory model under the synchronous scheduler, where the algorithm guarantees two issues. First, the convergence time complexity wrt. mutual exclusion is optimal. Second, the algorithm guarantees that starting from any configuration, after finite number of steps, the recurrence of granting a privilege to any process is 1.0; i.e. in each synchronous step, exactly one process is granted a privilege. Furthermore, this chapter presents algorithms showing the trade-off between the following complexity measures: the convergence time complexity wrt. mutual exclusion, the convergence time wrt. Δ recurrence of granting a privilege, and the space requirement. Some of this chapter's content has appeared in [3, 7] of the author's publications.

This chapter is structured as follows. Section 4.1 presents the mutual exclusion problem and related work. Section 4.2 explains the problem statement, the assumptions, and the contributions of this chapter. Next, Section 4.3 presents three mutual exclusion algorithms, showing trade-off between their complexity measures. Section 4.4 provides correctness proofs of the complexity measures of the algorithms. Next, Section 4.5 shows the optimality proof of the convergence time complexity wrt. mutual exclusion under the synchronous scheduler, which appears in two algorithms in Section 4.3. Finally, Section 4.6 presents concluding remarks.

4.1 The Mutual Exclusion Problem

Mutual exclusion – shortly *mutex* – is a significant problem in concurrent and distributed computing. Mutex is pioneered by Dijkstra in 1965 [Dij65]. It is also a significant topic in self-stabilization since 1974 [Dij74]. The following parts in this section present the mutex problem and related work.

4.1.1 Mutual Exclusion

Mutex is considered in systems having at least two processes performing concurrent tasks. Given a set of multiple processes running tasks concurrently, mutex comprises two properties. First, there exists a particular action that should not be executed, or enabled, by two processes simultaneously. Such an action is referred to as an *access to a critical section*, that should not be accessed by more than one process in each step. This property is a safety property. Second, each process must be enabled to access

the critical section infinitely often. This property is a liveness property, that is usually called *fairness*, because it ensures that no process is blocked forever to access the critical section.

There are two classical assumptions regarding mutex. First, the behavior of "accessing the critical section" by a process does not block another process to access the critical section. Second, no process may stay in the critical section forever.

The classical problem of mutex was first identified and addressed by Dijkstra in the seminal work of [Dij65]. Here, Dijkstra introduced a mutex algorithm executed by processes having access to shared variables. Each process executes the actions of the algorithm in a sequential order. With this algorithm, mutex is guaranteed under two assumptions. First, some variables must be initialized by some values before any process accesses the critical section for the first time. Second, there exist no transient faults. These two assumptions indicate that self-stabilization wrt. mutex is not concerned in this algorithm. Many other solutions followed Dijkstra's work [Dij65], such as Lamport's algorithm [Lam74], Peterson's algorithm [Pet81], and others [Ray86].

In the example above, failing to satisfy the safety property of mutex is critical: it may lead to having wrong outputs. Safety critical systems are required to guarantee mutex under all circumstances. However, for other systems, mutex is needed to guarantee the delivery of a certain quality of service, and is not safety critical. For example, if two parallel processes are responsible for scheduling, prioritizing, or sorting entities stored in a shared resource, then failing to satisfy mutex for a finite time may only delay the sorting operation. This is usually not as critical as providing wrong outputs. Self-stabilization wrt. mutex in this case is useful to overcome failures due to transient faults in sorting. In relation to this, the pioneering work of self-stabilization by Dijkstra [Dij74] introduced the first self-stabilizing distributed mutex algorithms for ring topologies that use the shared memory model, in which there are no globally shared variables; i.e. the variables of a process are read by only the process itself and its neighbors.

This chapter addresses self-stabilization wrt. mutex in the sense of [Dij74], i.e. when a system is required to be self-stabilizing wrt. mutex.

4.1.2 Related Work

The early work concerning self-stabilization in spite of distributed control, which has been pioneered by Dijkstra [Dij74], introduces three self-stabilizing mutex algorithms for ring topologies using the shared memory model. The algorithms converge wrt. mutex in $\mathcal{O}(\mathsf{n}^2)$ steps under the asynchronous scheduler [CSZ10], where n is the number of processes. The three algorithms require three, four, and n values for each process, respectively. A following work of Lamport considers mutex and its self-stabilization in [Lam86]. This work takes into consideration many real-world requirements other than the basic ones. For example, it considers prioritizing the actions by first-requested first-served. It also considers having levels of the fairness requirement. Later, in [DIM93], the authors present a self-stabilizing mutex algorithm for connected graphs using the shared memory model, where the real/write operations are not atomic. The algorithm is based on building a spanning tree through the graph, and passing a token in a depth-first manner through the tree. The convergence time complexity is $\mathcal{O}(\mathsf{n} \cdot \mathsf{diam}(\mathcal{G}))$ under the synchronous scheduler.

A later work extends mutex by many other properties. The first property is the *group mutex* [Jou98]: for this property, it is assumed that there are many critical sections that may have different types. The basic requirement is that if two processes access two different critical sections, then both accessed critical sections must have the same type. Examples of self-stabilizing algorithms covering this problem are [CP00, BB11]. The second property is the *local mutex* [HP92]: the requirement is that if a process is in the critical section, then none of the process's neighbors is also in the critical section, but other processes are allowed to do so. Local mutex reflects the well-known Dining Philosophers' problem. Examples of self-stabilizing solutions wrt. this property are [KY02, BDGM02]. Furthermore, there is the *ℓ-exclusion* problem [FLBB79]: the requirement is that at most ℓ processes can simultaneously access the critical section. Examples of self-stabilizing solutions wrt. this property are [DDHL09, CDDL15].

In a recent work, Dubois et al. [DG13] exploit a particular approach to design a self-stabilizing mutex algorithm for connected graph using the shared memory model. The approach basically pertains the so-called *phase clock* or *unison* [GH90, CFG92, BPV04, BPV08]. In a phase clock system, each process has an integer variable – a clock – that is incremented within a range of values. The requirement of such a system is to keep a bounded difference between the clocks belonging to all processes. Mutex is achieved as follows: each process is assigned a constant value, such that the difference between the constant values assigned to any two processes is greater than the difference between the clock values belonging to the processes. Next, a process is granted a privilege to access the critical section only if its clock value is equal to its constant value. The significance of the approach [DG13] is that the convergence time complexity is $\lceil \mathsf{diam}(\mathcal{G})/2 \rceil$ under the synchronous scheduler, which has not been achieved before. This complexity is claimed to be optimal in [DG13]. Section 4.5 of this work points out a flaw in the optimality proof of [DG13] and refines it.

The performance of self-stabilizing mutex algorithms has many aspects: it considers the convergence time, e.g. [CSZ10], the space requirement, e.g. [BPV04], the generality of the topology, e.g. [HMR94], the scheduler, e.g. [DGT00], and other aspects. Measuring the recurrence of granting a privilege to processes is usually referred to as *service time*. This aspect is considered explicitly for unidirectional anonymous ring topologies in [Joh02, Joh04], in which a single token is circulated through processes in a ring. A process is, then, granted a privilege if it owns the token. In [Joh02], the author provides an algorithm that once has stabilized, it provides optimal service time; i.e. for each n steps, each process obtains the single token once. The space requirement is optimized for the same problem in [Joh04]. In comparison to Dijkstra's approach [Dij74], the algorithms of [Joh02, Joh04] do not assume the existence of process ids, nor an identified process that executes a different sub-algorithm

Considering the recurrence of granting a privilege – in other words: the service time – in connected graphs under the synchronous scheduler, there exists no approach that provides optimal recurrence of granting a privilege – which equals the recurrence of 1.0 – together with optimal convergence time wrt. mutex. The approach of [BPV08] can be applied straightforward to achieve optimal recurrence of granting a privilege. However, the convergence time wrt. mutex using [BPV08] is $\mathsf{diam}(\mathcal{G})-1$. In addition, the approach of [DG13], which uses the asynchronous unison [BPV04], does not provide optimal recurrence of granting a privilege, even when using the synchronous unison [BPV08].

4.2 Problem Statement and Assumptions

In this section, the aim is to design an efficient self-stabilizing mutex algorithm, with a focus on the recurrence of granting a privilege to processes.

Recall that the specification of mutex is that, in each configuration, at most one process is privileged to access the critical section, and each process is privileged infinitely often. This specification is formalized in the following definition.

Definition 4.1 (Mutual Exclusion – ME). Let $\mathcal{G} = (\mathcal{P}, \mathcal{E})$ be a graph where $\mathcal{P}$ is a set of processes, let A be an algorithm, and let *privileged* be a condition on each state of each process $p \in \mathcal{P}$ (or shortly $privileged_p$). A specification ME, denoting *mutual exclusion* (wrt. *privileged*), is said to be satisfied by an execution $\Xi : \gamma_0, \gamma_1, \dots$ of A over $\mathcal{G}$ iff:

1. Safety: if *privileged* holds for a process $p \in \mathcal{P}$ in a configuration γ_i, then for each process $q \in \mathcal{P} \setminus \{p\}$, $privileged_q$ does not hold in γ_i.
2. Liveness: in Ξ, *privileged* holds for each process $p \in \mathcal{P}$ infinitely often. $\diamondsuit$

4.2.1 Problem Statement

The aim is to design a self-stabilizing mutex algorithm for connected graphs using the shared memory model and working under the synchronous scheduler, such that the algorithm has an optimal convergence time wrt. mutex, and reaches optimal recurrence of granting a privilege to processes; i.e. a recurrence of 1.0.

Problem 4.1. Devise a self-stabilizing distributed mutual exclusion algorithm for connected graphs using the shared memory model under the synchronous scheduler, such that

1. The algorithm has an optimal convergence time wrt. mutex.
2. The algorithm converges to an optimal recurrence of granting unique privilege. $\diamondsuit$

The contributions of this chapter are as follows:

- A self-stabilizing mutex algorithm for connected graphs using the shared memory model under the synchronous scheduler is presented. The algorithm is based on an extended version of the synchronous unison [BPV08]. The convergence time wrt. mutex is $\lceil \mathsf{diam}(\mathcal{G})/2 \rceil - 1$, which is optimal. The algorithm converges to 1.0 recurrence of granting a privilege in $\lceil 2.5 \cdot \mathsf{diam}(\mathcal{G}) \rceil - 1$ synchronous steps.
- Two other self-stabilizing mutex algorithms based on the synchronous unison [BPV08] are presented. The algorithms, together with the first one, show the trade-off between the convergence time wrt. mutex, the convergence time wrt. Δ recurrence of granting a privilege, the value of Δ, and the space requirements.
- The optimality of the convergence time wrt. mutex, that equals $\lceil \mathsf{diam}(\mathcal{G})/2 \rceil - 1$, is justified by refining the earlier proof of [DG13], which – wrongly – claims the optimality of $\lceil \mathsf{diam}(\mathcal{G})/2 \rceil$.

4.2.2 Assumptions

The assumptions in this chapter are as follows:

- The topology is a connected graph $\mathcal{G} = (\mathcal{P}, \mathcal{E})$, where $\mathcal{P}$ is a set of *processes*.
- The communication model used by any topology is the *shared memory model*, and the used scheduler is the *synchronous scheduler*.
- The number of processes, n, and the diameter of the topology, $\mathsf{diam}(\mathcal{G})$, are known parameters by each process.

4.3 Mutual Exclusion Algorithms

In this section, three self-stabilizing mutex algorithms, showing trade-off between complexity measures, are presented. Two algorithms are based on the basic synchronous unison algorithm given by Algorithm 3 of [BPV08]. The third one is based on an extended version of Algorithm 3 of [BPV08]. Section 4.3.1 shows the synchronous unison algorithm of [BPV08], and Section 4.3.2 presents the mutex algorithms.

4.3.1 Finite Incrementing System

Algorithm 3 of [BPV08] implements a synchronous incrementing system with a finite domain of values. The aim of this system is to repeatedly increment a value belonging to each process, in each synchronous step, while keeping the values of all processes equal. The values are required to be equal only under the synchronous scheduler. Under the asynchronous scheduler, this is not possible, and instead, a bounded difference between the values is required to be preserved. The finite incrementing system is shown in Figure 4.1. The system's parameters are as follows:

- two natural numbers $\mathcal{K}, \alpha \in \mathbb{N}_0$. The number $\mathcal{K}$ will later represent an upper bound on the domain of the incrementing system, and the number $-\alpha$ will define a lower bound.
- a finite set $\mathcal{X} = \{-\alpha, \ldots, 0, \ldots, \mathcal{K}{-}1\}$
  - a subset $\mathsf{tail}_{\mathcal{X}} = \{-\alpha, \ldots, 0\}$
  - a subset $\mathsf{tail}^*_{\mathcal{X}} = \mathsf{tail}_{\mathcal{X}} \setminus \{0\}$
  - a subset $\mathsf{stab}_{\mathcal{X}} = \{0, \ldots, \mathcal{K}{-}1\}$
- a function $\varphi : \mathcal{X} \longrightarrow \mathcal{X}$, where $\varphi(a) = (a + 1 \bmod \mathcal{K})$ if $a \geq 0$, and $\varphi(a) = a + 1$ otherwise

The pair $(\mathcal{X}, \varphi)$ is called a *finite incrementing system*, whose domain is $\mathcal{X}$, and its incrementing function is φ. Note that the value of $\varphi(\mathcal{K}-1)$ is 0, which is also considered an incrementation for having a finite domain.

This system is applied to connected graphs under the synchronous scheduler. Each process p has one variable r_p, whose domain is $\mathcal{X}$. The aim of this system is twofold: first to keep the values of r equal among all processes, and second, to increment the value of r_p in each step. This aim is the specification of the so-called *synchronous unison*.

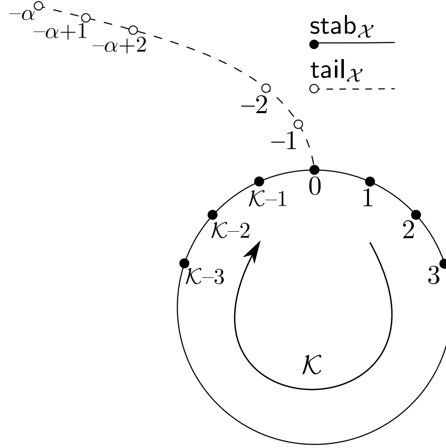


Figure 4.1: Finite incrementing system with $\alpha < 0$ and $\mathcal{K} > 0$

Definition 4.2 (Synchronous Unison – SU). Let $\mathcal{G} = (\mathcal{P}, \mathcal{E})$ be a topology, and let A be an algorithm. A specification SU, denoting *synchronous unison* (wrt. r), is said to be satisfied by an execution $\gamma_0, \gamma_1, \ldots$ of A over $\mathcal{G}$ iff:

1. Safety: in each configuration $\gamma_0, \gamma_1, \ldots$, for all $p, q \in \mathcal{P}$, $\mathsf{r}_p = \mathsf{r}_q$.
2. Liveness: each process $p \in \mathcal{P}$ executes $\varphi(\mathsf{r}_p)$ in each step of Ξ. ♢

The finite incrementing system is obtained by a distributed algorithm, where each process p has three guarded commands, explained as follows (recall that $\mathcal{N}_p$ denotes the set of neighbors of a process p, and $\mathcal{N}_p^*$ denotes $\{p\} \cup \mathcal{N}_p$):

1. *NA* (*Normal Action*): If for all $q \in \mathcal{N}_p$, $\mathsf{r}_p = \mathsf{r}_q$ and $p \in \mathsf{stab}_\mathcal{X}$ hold, then p increments r_p within $\mathsf{stab}_\mathcal{X}$.
2. *CA* (*Converge Action*): if there exists $q \in \mathcal{N}_p^*$ such that $\mathsf{r}_q \in \mathsf{tail}_\mathcal{X}^*$, then r_p is set to $\mathsf{r}_g + 1$, where $g \in \mathcal{N}_p^*$ is the process with the minimum value of r in $\mathcal{N}_p^*$.
3. *RA* (*Reset Action*): if for all $q \in \mathcal{N}_p^*$, $\mathsf{r}_q \in \mathsf{stab}_\mathcal{X}$, and there exists $g \in \mathcal{N}_p$ such that $\mathsf{r}_g \neq \mathsf{r}_p$, then r_p is reset to $-\alpha$.

The commands are illustrated using Figure 4.1 as follows: Let p be a process. The command *NA* is executed when the value of r_p is within the big circle, and equals the values of r for all neighbors. This denotes the normal behavior of the process. Next, the command *CA* is executed when the value of r_p is in the dashed area; i.e. during the convergence wrt. SU. Finally, the command *RA* is executed when the value of r_p is within the big circle, and one neighbor has a different value of r. This denotes a moment when p detects a failure and resets its value.

The results of [BPV08] state that, choosing $\alpha \geq \mathsf{diam}(\mathcal{G})$, and $\mathcal{K} \geq 2$, implies that the finite incrementing system is self-stabilizing wrt. SU in $2 \cdot \mathsf{diam}(\mathcal{G})$ steps. The intuition behind these choices is as follows: α is required to be large enough, to be able to execute *CA* for enough subsequent steps to equalize the values of r for all processes without any reset. The choice of $\mathcal{K}$ should ensure that r is incremented infinitely often.

Note that, for completeness, because the incrementing system of [BPV08] is modified for one algorithm in the following section, the correctness proofs in Section 4.4 cover all properties regardless of the mentioned results of [BPV08].

4.3.2 Algorithms

The finite incrementing system is used to design three self-stabilizing mutex algorithms. This approach is inspired from [BPV04, DG13]. The basic idea is as follows:

- The condition $privileged_p$ denotes that a process p has a privilege. It is defined as a predicate over the value of r_p and the id of p, in a way that if the value of r is equal among all processes, the safety property of ME holds.
- The algorithms are self-stabilizing wrt. SU, which implies that the algorithms are self-stabilizing wrt. the safety property of ME.
- The value $\mathcal{K}$ is set to be greater or equal to n. Next, for each process p, there exists a unique value v in $\{0, \dots, \mathcal{K}-1\}$, such that p is privileged if $\mathsf{r}_p = v$. Once each of the algorithms stabilizes, after a finite number of steps, the function φ increments the values of r within $\mathsf{stab}_\mathcal{X} = \{0, \dots, \mathcal{K}-1\}$ repeatedly, which grants each process a privilege infinitely often. This preserves the liveness property of ME.

The following sections present the algorithms.

Algorithm 4.1

Algorithm 4.1 uses the classical finite incrementing system defined in Section 4.3.1. The values of α and $\mathcal{K}$ are set as follows: $\alpha = \mathsf{diam}(\mathcal{G})$ and $\mathcal{K} = \mathsf{n}$. Given that $\mathsf{n} \geq 2$, Algorithm 4.1 is self-stabilizing wrt. SU in $2 \cdot \mathsf{diam}(\mathcal{G})$ steps. Since $\mathcal{K} = \mathsf{n}$, each value in $\mathsf{stab}_\mathcal{X}$ corresponds to a process id, and vice versa. A process p_i is privileged iff two conditions hold: (1) $\mathsf{r}_i = i$, and (2) $\mathsf{r}_i = \mathsf{r}_j$ for each $p_j \in \mathcal{N}_{p_i}$. The values of r, in which a process may be privileged, are marked by black circles in the figure of Algorithm 4.1. The other values are marked by white circles. By this setting, since the incrementing system converges to an execution satisfying SU and increments the value of r of each process within $\{0, \dots, \mathsf{n}-1\}$, it follows that ME holds.

Since each element of $\mathsf{stab}_\mathcal{X}$ corresponds to a process id, there exists exactly one privileged process in each configuration. This implies that Algorithm 4.1 achieves $\Delta = 1.0$ recurrence of granting a privilege.

Algorithm 4.1 converges to ME in at most $\mathsf{diam}(\mathcal{G})-1$ steps. In Figure 4.2, an example shows that this convergence time complexity is a lower bound for Algorithm 4.1. The example shows a bus topology $\mathcal{G}$ consisting of 6 processes, and having ids $\{0, \dots, 5\}$, where $\mathsf{diam}(\mathcal{G}) = 5$. The initial configuration is γ_0. In the step (γ_0, γ_1), all processes execute CA. In (γ_1, γ_2), p_1, p_3, p_5 execute CA, and the others execute NA. In (γ_2, γ_3), p_1, p_2, p_3, p_4 execute NA, and p_0, p_5 execute RA. In γ_3, p_1 and p_2 are privileged, which indicates that the safety property of ME does not hold. This implies that ME is not guaranteed in $3 = \mathsf{diam}(\mathcal{G})-2$ steps. Note that after $\mathsf{diam}(\mathcal{G})-1$ steps, in γ_4, ME holds. Section 4.4 provides a correctness proof about the convergence time complexity wrt. ME.

The following section presents Algorithm 4.2.

Algorithm 4.1 Mutual exclusion algorithm based on the classical finite incrementing system

---

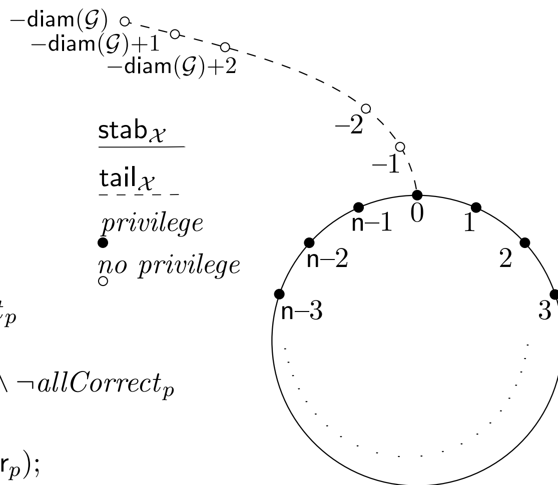
// This represents the sub-algorithm for each process p

Constants

$\mathcal{K}$	$=$	n
α	$=$	$\mathsf{diam}(\mathcal{G})$
$\mathsf{stab}_{\mathcal{X}}$	$=$	$\{0, \ldots, \mathcal{K}-1\}$
$\mathsf{tail}_{\mathcal{X}}$	$=$	$\{-\alpha, \ldots, 0\}$
$\mathsf{tail}^*_{\mathcal{X}}$	$=$	$\mathsf{tail}_{\mathcal{X}} \setminus \{0\}$

Predicates

$allCorrect_p$	$\equiv$	$\forall\, q \in \mathcal{N}_p \bullet \mathsf{r}_p = \mathsf{r}_q$
$privileged_p$	$\equiv$	$allCorrect_p \wedge \mathsf{r}_p = \mathsf{id}$
$normal_p$	$\equiv$	$\mathsf{r}_p \in \mathsf{stab}_{\mathcal{X}} \wedge allCorrect_p$
$converge_p$	$\equiv$	$\exists\, q \in \mathcal{N}^*_p \bullet \mathsf{r}_q \in \mathsf{tail}^*_{\mathcal{X}}$
$reset_p$	$\equiv$	$\forall\, q \in \mathcal{N}^*_p \bullet \mathsf{r}_q \in \mathsf{stab}_{\mathcal{X}} \wedge \neg allCorrect_p$



Guarded Commands

NA	::	$normal_p$	$\longrightarrow$	$\mathsf{r}_p := \varphi(\mathsf{r}_p)$;
CA	::	$converge_p$	$\longrightarrow$	$\mathsf{r}_p := \mathsf{min}\{\varphi(\mathsf{r}_q) \mid q \in \mathcal{N}^*_p\}$;
RA	::	$reset_p$	$\longrightarrow$	$\mathsf{r}_p := -\alpha$;

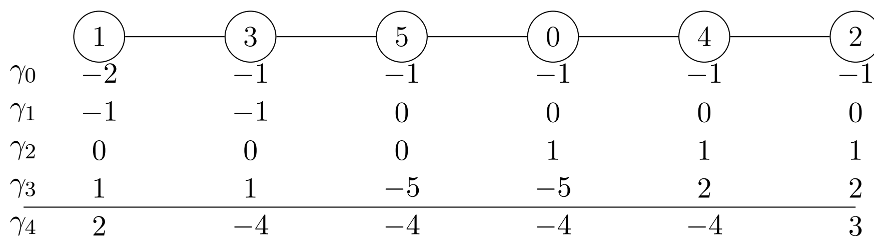


Figure 4.2: Execution of Algorithm 4.1 over a topology with $\mathsf{n} = 6$, and $\mathsf{diam}(\mathcal{G}) = 5$. The processes p_1 and p_2 are privileged in γ_3, i.e., after $3 = \mathsf{diam}(\mathcal{G}) - 2$ steps.

Algorithm 4.2

Algorithm 4.2 has a significance over Algorithm 4.1 that the convergence time of Algorithm 4.2 to ME is $\lceil \mathsf{diam}(\mathcal{G})/2 \rceil - 1$. For convenient presentation, the value $\lceil \mathsf{diam}(\mathcal{G})/2 \rceil - 1$ is denoted by ϵ in some positions.

This convergence time is achieved by re-defining the set $\mathsf{stab}_{\mathcal{X}}$, such that scenarios similar to the one in Figure 4.2 are avoided. In detail, first, $\mathsf{stab}_{\mathcal{X}}$ is defined to include $n + \epsilon$ values. Next, it is asserted that no process p is privileged if r_p has a value from $\{0, \ldots, \epsilon{-}1\}$; i.e. a processes may be privileged only if it has a value in $\{\epsilon, \ldots, \mathsf{n}{+}\epsilon{-}1\}$. With this setting, after the convergence wrt. SU by executing *CA*, there exists a gap in which the values of r are in $\mathsf{stab}_{\mathcal{X}}$ but no process is privileged. This prevents the scenarios where mutex is violated in the configurations $\gamma_\epsilon, \ldots, \gamma_{\mathsf{diam}(\mathcal{G})-2}$ by Algorithm 4.1.

Since *NA* increments r within the values $\{0, \ldots, \mathsf{n}{+}\epsilon{-}1\}$, and since no process p is privileged if $\mathsf{r}_p \in \{0, \ldots, \epsilon{-}1\}$, the recurrence of *privileged* is less than 1.0 if $\mathsf{diam}(\mathcal{G}) > 2$. In fact, Algorithm 4.2 guarantees that the recurrence of *privileged* is at least $\mathsf{n}/(\mathsf{n} + \epsilon)$.

Algorithm 4.2 Mutual exclusion algorithm with optimal ME-convergence time $\lceil \mathsf{diam}(\mathcal{G})/2 \rceil - 1$

// This represents the sub-algorithm for each process p

Constants

$\epsilon \quad = \quad \lceil \mathsf{diam}(\mathcal{G})/2 \rceil - 1$

$\mathcal{K} \quad = \quad \mathsf{n} + \epsilon$

$\alpha \quad = \quad \mathsf{diam}(\mathcal{G})$

$\mathsf{stab}_{\mathcal{X}} \quad = \quad \{0, \ldots, \mathcal{K} - 1\}$

$\mathsf{tail}_{\mathcal{X}} \quad = \quad \{-\alpha, \ldots, 0\}$

$\mathsf{tail}^*_{\mathcal{X}} \quad = \quad \mathsf{tail}_{\mathcal{X}} \setminus \{0\}$

Predicates

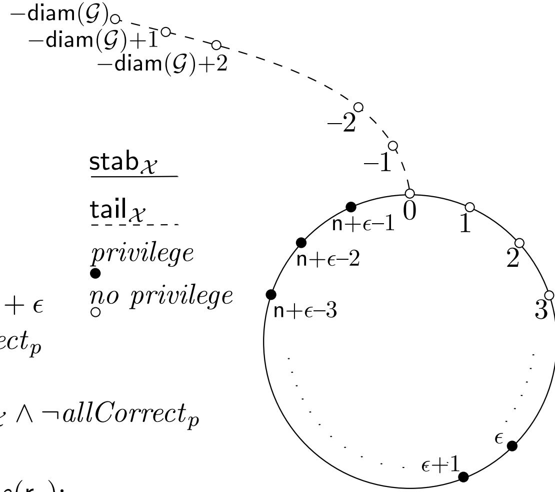
$allCorrect_p \quad \equiv \quad \forall\, q \in \mathcal{N}_p \bullet \mathsf{r}_p = \mathsf{r}_q$

$privileged_p \quad \equiv \quad allCorrect_p \wedge \mathsf{r}_p = \mathsf{id} + \epsilon$

$normal_p \quad \equiv \quad \mathsf{r}_p \in \mathsf{stab}_{\mathcal{X}} \wedge allCorrect_p$

$converge_p \quad \equiv \quad \exists\, q \in \mathcal{N}^*_p \bullet \mathsf{r}_q \in \mathsf{tail}^*_{\mathcal{X}}$

$reset_p \quad \equiv \quad \forall\, q \in \mathcal{N}^*_p \bullet \mathsf{r}_q \in \mathsf{stab}_{\mathcal{X}} \wedge \neg allCorrect_p$



Guarded Commands

NA :: $normal_p \longrightarrow \mathsf{r}_p := \varphi(\mathsf{r}_p)$;

CA :: $converge_p \longrightarrow \mathsf{r}_p := \mathsf{min}\{\varphi(\mathsf{r}_q) \mid q \in \mathcal{N}^*_p\}$;

RA :: $reset_p \longrightarrow \mathsf{r}_p := -\alpha$;

Algorithms 4.1 and 4.2 are based on the finite incrementing system that achieves synchronous unison, given by Algorithm 3 of [BPV08]. Applying this finite incrementing system with the ideas of designing mutex algorithms does not achieve both (1) optimal convergence time wrt. mutex, and (2) convergence wrt. 1.0 recurrence of granting a privilege. For this reason, in the following section, the finite incrementing system is re-engineered to achieve the goal.

Algorithm 4.3

The finite incrementing system is re-engineered as follows (cf. Algorithm 4.3): a subset of $\mathsf{stab}_{\mathcal{X}}$ is defined to be out of the big circle, and it is set that no process is privileged if its value of r belongs to this subset. This subset guarantees the optimal convergence time wrt. ME, similar to Algorithm 4.2. Next, the function φ finally increments the values of r within only the big circle whose values correspond to all processes' ids. This guarantees achieving 1.0 recurrence of *privileged*.

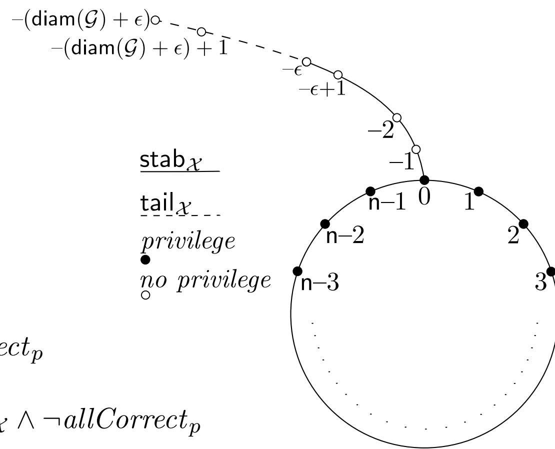
In detail, first, the sets $\mathsf{tail}_{\mathcal{X}}$ and $\mathsf{stab}_{\mathcal{X}}$ are defined as $\mathsf{tail}_{\mathcal{X}} = \{-(\mathsf{diam}(\mathcal{G})+\epsilon), \ldots, -\epsilon\}$, and $\mathsf{stab}_{\mathcal{X}} = \{-\epsilon, \ldots, 0, \ldots, \mathsf{n}-1\}$. Second, it is set that no process p may be privileged if r_p has a value from $\{-\epsilon, \ldots, -1\}$, and p may be privileged if r_p is within $\{0, \ldots, \mathsf{n}-1\}$. Next, after a finite number of steps of an execution, the command *NA* increments the variables only within $\{0, \ldots, \mathsf{n}-1\}$ to guarantee 1.0 recurrence of *privileged*.

Algorithm 4.3 Mutual exclusion algorithm with optimal ME-convergence time, and achieving 1.0 recurrence of *privileged*

// This represents the sub-algorithm for each process p

Constants

$\epsilon = \lceil \mathsf{diam}(\mathcal{G})/2 \rceil - 1$
$\mathcal{K} = \mathsf{n}$
$\alpha = \mathsf{diam}(\mathcal{G}) + \epsilon$
$\mathsf{stab}_{\mathcal{X}} = \{-\epsilon, \ldots, 0, \ldots, \mathcal{K} - 1\}$
$\mathsf{tail}_{\mathcal{X}} = \{-\alpha, \ldots, -\epsilon\}$
$\mathsf{tail}^*_{\mathcal{X}} = \mathsf{tail}_{\mathcal{X}} \setminus \{-\epsilon\}$



Predicates

$allCorrect_p \equiv \forall q \in \mathcal{N}_p \bullet \mathsf{r}_p = \mathsf{r}_q$
$privileged_p \equiv \mathsf{r}_p = \mathsf{id} \wedge allCorrect_p$
$normal_p \equiv \mathsf{r}_p \in \mathsf{stab}_{\mathcal{X}} \wedge allCorrect_p$
$converge_p \equiv \exists q \in \mathcal{N}^*_p \bullet \mathsf{r}_q \in \mathsf{tail}^*_{\mathcal{X}}$
$reset_p \equiv \forall q \in \mathcal{N}^*_p \bullet \mathsf{r}_q \in \mathsf{stab}_{\mathcal{X}} \wedge \neg allCorrect_p$

Guarded Commands

$NA :: normal_p \longrightarrow \mathsf{r}_p := \varphi(\mathsf{r}_p);$
$CA :: converge_p \longrightarrow \mathsf{r}_p := \mathsf{min}\{\varphi(\mathsf{r}_q) \mid q \in \mathcal{N}^*_p\};$
$RA :: reset_p \longrightarrow \mathsf{r}_p := -\alpha;$

4.4 Correctness and Time Complexity

This section presents the correctness and time complexity proofs for Algorithms 4.1–4.3. First, Section 4.4.1 addresses the property SU. Next, Section 4.4.2 addresses the property ME. After that, Section 4.4.3 addresses recurrence properties.

For convenient presentation, some notation is presented as follows: for each process p_j, the notation r^i_j as well as $\mathsf{r}^i_{p_j}$ is used to denote the value of r_j in a configuration γ_i.

4.4.1 Self-Stabilization wrt. SU

It is shown that Algorithms 4.1–4.3 are self-stabilizing wrt. SU in $2 \cdot \mathsf{diam}(\mathcal{G})$ steps. Note that this holds for Algorithms 4.1 and 4.2 following Corollary 3 of [BPV08]. However, the proofs are re-constructed to cover Algorithm 4.3 and the other properties in the following parts.

The following lemmata in this section apply to Algorithms 4.1–4.3. It is assumed that there is a graph $\mathcal{G} = (\mathcal{P}, \mathcal{E})$ using the shared memory model under the synchronous scheduler.

The following lemma shows the closure of SU.

Lemma 4.1 (SU Closure)**.** If SU holds in a configuration γ_i, then SU holds also in γ_{i+1}.

Proof. Let γ_i be a configuration in which SU holds. By Definition 4.2, for all $p, q \in \mathcal{P}$, $\mathsf{r}_p^i = \mathsf{r}_q^i$. Two cases are distinguished:

1. In the first case, $\mathsf{r}_p^i \in \mathsf{tail}_{\mathcal{X}}^*$. In the step (γ_i, γ_{i+1}), each process executes CA, implying that $\mathsf{r}_p^{i+1} = \mathsf{r}_q^{i+1}$.
2. In the second case, $\mathsf{r}_p^i \in \mathsf{stab}_{\mathcal{X}}$. In the step (γ_i, γ_{i+1}), each process executes NA, implying that $\mathsf{r}_p^{i+1} = \mathsf{r}_q^{i+1}$.

In both cases, the safety property of SU holds in γ_{i+1}. If the safety property holds in a configuration, then each process has exactly one enabled command from NA and CA, and the command RA is not enabled for any process. Each of the possibly enabled commands increments the value of its corresponding process. The liveness property of SU holds. □

The following two lemmata and the corollary concern the command RA.

Lemma 4.2. Given an execution $\gamma_0, \gamma_1, \ldots,$ if no process executes RA within $(\gamma_0, \gamma_1), \ldots, (\gamma_{\mathsf{diam}(\mathcal{G})-1}, \gamma_{\mathsf{diam}(\mathcal{G})})$, then SU holds in $\gamma_{\mathsf{diam}(\mathcal{G})}$.

Proof. Let $(\gamma_0, \gamma_1), \ldots, (\gamma_{\mathsf{diam}(\mathcal{G})-1}, \gamma_{\mathsf{diam}(\mathcal{G})})$ be an execution, within which no process executes RA. If SU holds in any configuration γ_e for $e \in \{0, \ldots, \gamma_{\mathsf{diam}(\mathcal{G})-1}\}$, then by Lemma 4.1 SU holds in all $\gamma_{e+1}, \gamma_{e+2}, \ldots,$ which implies the claim. Assume that SU does not hold in γ_0, and let $p_i \in \mathcal{P}$ be a process where r_i^0 is the minimum among all processes. Based on the value of r_i^0, two cases are distinguished:

1. $\mathsf{r}_i^0 \in \mathsf{stab}_{\mathcal{X}}$ holds. In this case, by assumption, for each process $p' \in \mathcal{P} \setminus \{p_i\}$, $p' \in \mathsf{stab}_{\mathcal{X}}$ holds. Since γ_0 does not satisfy SU, there exist two neighboring processes p_a, p_b, such that $\mathsf{r}_a^0 \neq \mathsf{r}_b^0$, and $\mathsf{r}_a^0, \mathsf{r}_b^0 \geq \mathsf{min}(\mathsf{stab}_{\mathcal{X}})$. By definition, p_a and p_b execute RA in (γ_0, γ_1), which contradicts the assumption. This case does not exist.
2. $\mathsf{r}_i^0 \in \mathsf{tail}_{\mathcal{X}}^*$ holds. Given that no process executes RA in (γ_0, γ_1) by assumption, p_i executes either NA or CA, implying that $\mathsf{r}_i^1 = \mathsf{r}_i^0 + 1$. If $\mathsf{r}_i^1 \in \mathsf{stab}_{\mathcal{X}}$, then SU holds, following the argument of the first case. Otherwise, $\mathsf{r}_i^1 \in \mathsf{tail}_{\mathcal{X}}^*$, and by definition of CA and NA, each process $p_j \in \mathcal{N}_{p_i}$ executes CA in (γ_0, γ_1), and therefore, $\mathsf{r}_j^1 = \mathsf{r}_i^1$. All other processes $p_s \notin \mathcal{N}_{p_i}^*$ execute either CA or NA. By definition of these commands, and since r_i^0 is the minimum, it follows that $\mathsf{r}_s^1 \geq \mathsf{r}_i^1$. Now all processes

at distance 1 of p_i have the same value of r_i^1, and r_i^1 is the minimum. Inductively, the processes at distance 2, 3,... of p_i perform similar steps. Since the distance between p_i and any process is bounded by $\mathsf{diam}(\mathcal{G})$, all processes have the same value of r_i after $\mathsf{diam}(\mathcal{G})$ steps.

This implies the safety property of SU in $\gamma_{\mathsf{diam}(\mathcal{G})}$. This, by Lemma 4.1, implies the liveness property of SU. □

Lemma 4.3. For each execution $\gamma_0, \gamma_1, \ldots,$ in each of the configurations $\gamma_{\mathsf{diam}(\mathcal{G})}, \gamma_{\mathsf{diam}(\mathcal{G})+1}, \ldots,$ for all $p, q \in \mathcal{P}$, if $\mathsf{r}_p, \mathsf{r}_q \in \mathsf{stab}_{\mathcal{X}}$, then $\mathsf{r}_p = \mathsf{r}_q$.

Proof. If no process executes RA within $\gamma_0, \ldots, \gamma_{\mathsf{diam}(\mathcal{G})}$, then by Lemma 4.2, SU holds in $\gamma_{\mathsf{diam}(\mathcal{G})}$, which by Lemma 4.1 implies that SU holds in $\gamma_{\mathsf{diam}(\mathcal{G})}, \gamma_{\mathsf{diam}(\mathcal{G})+1}, \ldots$. This implies the claim.

Assume that there exists a process p which executes RA in a step (γ_{i-1}, γ_i) for $(1 \leq i \leq \mathsf{diam}(\mathcal{G}))$. The proof is based on two arguments.

First argument: By definition of RA, $\mathsf{r}_p^i = -\alpha$ holds. Since each command may add at most 1 to the value of r_p, and $\alpha - i < \mathsf{diam}(\mathcal{G})$ by assumption, it follows that $\mathsf{r}_p^{\mathsf{diam}(\mathcal{G})} < \mathsf{min}(\mathsf{stab}_{\mathcal{X}})$ holds. In the step (γ_i, γ_{i+1}), each $q \in \mathcal{N}_p$ may execute only CA. By definition of CA, $\mathsf{r}_q^{i+1} \leq \mathsf{r}_p^{i+1}$ holds. Inductively, for each process g within a distance $\mathsf{diam}(\mathcal{G}) - i$ of p, $\mathsf{r}_g^{\mathsf{diam}(\mathcal{G})} \leq \mathsf{r}_p^{\mathsf{diam}(\mathcal{G})} < \mathsf{min}(\mathsf{stab}_{\mathcal{X}})$ holds.

Second argument: let p be a process, such that r_p^0 is the minimum among all other processes in $\mathcal{P}$. Let $v = \mathsf{r}_p^0$. By definition of the commands, if p executes either CA or NA, then $\mathsf{r}_p^1 = \varphi(v)$. If p executes RA, then by the first argument, $\mathsf{r}_p^{\mathsf{diam}(\mathcal{G})} < \mathsf{min}(\mathsf{stab}_{\mathcal{X}})$. Since r_p is the minimum value in γ_0, by definition of the commands:

- either for each p' within distance 1 of p, $\mathsf{r}_{p'}^1 = \varphi(v)$ holds, or
- $\mathsf{r}_{p'}^{\mathsf{diam}(\mathcal{G})} < \mathsf{min}(\mathsf{stab}_{\mathcal{X}})$ by the first argument.

In the next configuration γ_2, for each process p'' within distance 2 of p, by definition of the commands, either $\mathsf{r}_{p''}^2 = \varphi(\varphi(v))$ holds, or $\mathsf{r}_{p''}^{\mathsf{diam}(\mathcal{G})} < \mathsf{min}(\mathsf{stab}_{\mathcal{X}})$ holds, similar to the previous case. Inductively, in configuration $\gamma_{\mathsf{diam}(\mathcal{G})}$, for each process q within distance $\mathsf{diam}(\mathcal{G})$ of p (which covers all processes), either $\mathsf{r}_q^{\mathsf{diam}(\mathcal{G})} = \varphi^{\mathsf{diam}(\mathcal{G})}(v)$, or $\mathsf{r}_q^{\mathsf{diam}(\mathcal{G})} < \mathsf{min}(\mathsf{stab}_{\mathcal{X}})$. This implies that if there exists a process q where $\mathsf{r}_q^{\mathsf{diam}(\mathcal{G})} \in \mathsf{stab}_{\mathcal{X}}$, then $\mathsf{r}_q^{\mathsf{diam}(\mathcal{G})}$ is equal to $\varphi^{\mathsf{diam}(\mathcal{G})}(v)$.

The two arguments apply to any initial configuration, which implies the claim. □

Corollary 4.1. For each execution $\gamma_0, \gamma_1, \ldots,$ no process executes RA after $\gamma_{\mathsf{diam}(\mathcal{G})}$.

Proof. Follows from Lemma 4.3. □

The following lemma proves self-stabilization wrt. SU.

Lemma 4.4. Algorithms 4.1–4.3 are self-stabilizing wrt. SU in $2 \cdot \mathsf{diam}(\mathcal{G})$ steps.

Proof. For each execution $\gamma_0, \gamma_1, \ldots,$ of the algorithms, Corollary 4.1 implies that no process executes RA after $\gamma_{\mathsf{diam}(\mathcal{G})}$. Lemma 4.2 implies that SU holds if no process executes RA within $\mathsf{diam}(\mathcal{G})$ steps. This implies the claim. □

4.4.2 Self-Stabilization wrt. ME

This section concerns self-stabilization wrt. ME. It is shown that Algorithm 4.1 is self-stabilizing wrt. ME in $\mathsf{diam}(\mathcal{G})-1$ steps, and each Algorithm 4.2 and 4.3 is self-stabilizing wrt. ME in $\lceil \mathsf{diam}(\mathcal{G})/2\rceil -1$ steps.

The following lemma states that SU implies ME for the three algorithms.

Lemma 4.5. Given any Algorithm 4.1–4.3, if SU holds in a configuration γ_0, then ME holds in γ_0.

Proof. If SU holds in a configuration γ_0, then by Definition 4.2, $\forall\, p \in \mathcal{P}, q \in \mathcal{N}_p \bullet \mathsf{r}_p^0 = \mathsf{r}_q^0$. A process p is privileged only if $\mathsf{r}_p = \mathsf{id}_p$ (resp. $\mathsf{r}_p = \mathsf{id}_p + \epsilon$) by the definition of *privileged*, which implies that at most one process is privileged if SU holds (safety). By Lemma 4.1, for each execution $\gamma_0, \gamma_1, \ldots$, SU holds in all configurations $\gamma_1, \gamma_2, \ldots$. By definition of the commands, the possibly enabled commands for any process in any of the configurations $\gamma_0, \gamma_1, \ldots$ are *CA* and *NA*. Each of the commands increments r_p of each process p by (1 mod $\mathcal{K}$) in each step. By definition of φ, after finite steps, the values of r_p are incremented within $\{0, \ldots, \mathcal{K}-1\}$. By definition of *privileged*, for each process p, there exists a value v in $\{0, \ldots, \mathcal{K}-1\}$ such that, for each $q \in \mathcal{N}_p^*$, if $\mathsf{r}_q = v$ then p is privileged. It follows that the predicate *privileged* holds infinitely often for each process p (Liveness). □

The following theorem states the convergence time complexity for Algorithm 4.1.

Theorem 4.1. Algorithm 4.1 is self-stabilizing wrt. ME in $\mathsf{diam}(\mathcal{G})-1$ steps.

Proof. Since Algorithm 4.1 is self-stabilizing wrt. SU, and since – by Lemma 4.5 – ME holds if SU holds in any configuration, it follows that the liveness property of ME holds. Let $\gamma_0, \gamma_1, \ldots$ be an execution. It is shown that for all $i \geq \mathsf{diam}(\mathcal{G})-1$ steps, the safety property of ME holds in the configurations $\gamma_i, \gamma_{i+1}, \ldots$. Two cases are distinguished:

1. $i \geq \mathsf{diam}(\mathcal{G})$. By Lemma 4.3, for each execution $\gamma_0, \gamma_1, \ldots$, for each $p, q \in \mathcal{P}$, if $\mathsf{r}_p^{\mathsf{diam}(\mathcal{G})}, \mathsf{r}_q^{\mathsf{diam}(\mathcal{G})} \geq 0$, it follows that $\mathsf{r}_p^{\mathsf{diam}(\mathcal{G})} = \mathsf{r}_q^{\mathsf{diam}(\mathcal{G})}$. By definition of *privileged*, processes p, q can be privileged in $\gamma_{\mathsf{diam}(\mathcal{G})}$ only if $\mathsf{r}_p^{\mathsf{diam}(\mathcal{G})} \geq 0$ holds. By the uniqueness of the processes' ids, the safety property of ME holds in configurations $\gamma_i, \gamma_{i+1}, \ldots$.

2. $i = \mathsf{diam}(\mathcal{G})-1$. Assume by contradiction that there exist processes $p, q \in \mathcal{P}$, where both are privileged in configuration γ_i. This implies that $\mathsf{r}_p^i \neq \mathsf{r}_q^i$, and $\mathsf{r}_p^i, \mathsf{r}_q^i \geq 0$. By definition of privileged, for all $p' \in \mathcal{N}_p$, $\mathsf{r}_{p'}^i = \mathsf{r}_p^i$. The same applies to q. By definition of the commands, p and q execute *NA* in step (γ_i, γ_{i+1}). By definition of *NA*, $\mathsf{r}_p^{i+1} = \varphi(\mathsf{r}_p^i)$ holds. The same applies to q. This implies that $\mathsf{r}_p^{i+1} \neq \mathsf{r}_q^{i+1}$, and $\mathsf{r}_p^{i+1}, \mathsf{r}_q^{i+1} \geq 0$. By construction, $i{+}1 = \mathsf{diam}(\mathcal{G})$. This contradicts Lemma 4.3. □

Concerning the Algorithms 4.2 and 4.3, the convergence time complexity wrt. ME is shown to be equal to $\lceil \mathsf{diam}(\mathcal{G})/2\rceil -1$ steps. The proof idea is partially inspired from [DG13]: basically, the term *island* is borrowed from [DG13] and is extended to be used in the proofs. Note that it is not necessarily used the same way as in [DG13].

An island denotes a maximal strict subset of processes, whose values of r are equal, and are in $\mathsf{stab}_{\mathcal{X}}$. Clearly, if SU holds, then there exists no island. In addition, by the definition of *privileged* in Algorithms 4.1–4.3, at most one process may be privileged in any island.

Definition 4.3 (Island). Given a configuration γ_i of a topology $\mathcal{T} = (\mathcal{P}, \mathcal{E})$, an *island* is a maximal (wrt. inclusion) non-empty strict subset $I \subsetneq \mathcal{P}$, such that $\forall p, q \in I \bullet \mathsf{r}_p^i = \mathsf{r}_q^i \wedge \mathsf{r}_p^i \in \mathsf{stab}_{\mathcal{X}}$. ♢

The following definition specifies terms concerning the islands.

Definition 4.4. Let I be an Island in a topology $\mathcal{G}$.

- I is said to be an *init-island* iff $\forall p \in I \bullet \mathsf{r}_p = \mathsf{min}(\mathsf{stab}_{\mathcal{X}})$, and a *non-init-island* otherwise.
- The *border of I* ($\mathsf{Border}(I)$) is defined as follows: $\mathsf{Border}(I) = \{p \in I \mid \exists q \in \mathcal{P} \setminus I \bullet q \in \mathcal{N}_p\}$. The *depth* of I ($\mathsf{idepth}(I)$) is defined as follows: $\mathsf{idepth}(I) = \mathsf{max}\{\mathsf{min}\{\mathsf{dist}(\mathcal{G}, p, q) \mid q \in \mathsf{Border}(I)\} \mid p \in I\}$. ♢

The following two lemmata present properties of the islands.

Lemma 4.6. Let (γ_{i-1}, γ_i) be an execution step of Algorithm 4.2 or 4.3. If a process p belongs to a non-init-island I in γ_i, then p belongs to an island of depth $\mathsf{idepth}(I) + 1$ in γ_{i-1}.

Proof. Let p be a process belonging to a non-init-island I in γ_i. Assume, by contradiction, that p does not belong to any island in γ_{i-1}. This implies that $\mathsf{r}_p^{i-1} \in \mathsf{tail}_{\mathcal{X}}^*$ holds, implying that p may execute only CA in (γ_{i-1}, γ_i), and then $\mathsf{r}_p^i \in \mathsf{tail}_{\mathcal{X}}$ holds. By Definition 4.3, this implies that p either belongs to an init-island or to no island in γ_i. This contradicts that p belongs to a non-init-island in γ_i.

Let $\mathsf{idepth}(I) = h$. Assume, by contradiction, that p belongs to an island I' in γ_{i-1}, such that $\mathsf{idepth}(I') \neq h + 1$. By Definition 4.4, each $q \in \mathsf{Border}(I')$ executes RA or CA in (γ_{i-1}, γ_i). This decreases $\mathsf{idepth}(I')$ by 1 concerning that each process $g \in I' \setminus \mathsf{Border}(I')$ executes NA in (γ_{i-1}, γ_i) – by definition of the commands – implying that $I' \setminus \mathsf{Border}(I') = I$. This – following the assumption that $\mathsf{idepth}(I') \neq h+1$ – implies that the depth of I that contains p in γ_i is not equal to h. This contradicts the assumption that $\mathsf{idepth}(I) = h$. □

Lemma 4.7. Given Algorithm 4.2 or 4.3, in each configuration, there exists at most one island I such that $\mathsf{idepth}(I) \geq \lceil \frac{\mathsf{diam}(\mathcal{G})}{2} \rceil$.

Proof. Assume, by contradiction, that there exists a configuration with two different islands I and I', such that $\mathsf{idepth}(I), \mathsf{idepth}(I') \geq \lceil \frac{\mathsf{diam}(\mathcal{G})}{2} \rceil$. By Definition 4.4, there exists two processes $p \in I$, $p' \in I'$ such that: $\mathsf{min}\{\mathsf{dist}(p, q) \mid q \in \mathsf{Border}(I)\} \geq \lceil \frac{\mathsf{diam}(\mathcal{G})}{2} \rceil$, and $\mathsf{min}\{\mathsf{dist}(p', q') \mid q' \in \mathsf{Border}(I')\} \geq \lceil \frac{\mathsf{diam}(\mathcal{G})}{2} \rceil$. Let $path_1 : p, \ldots, p_i, \ldots, p'_i \ldots, p'$ be the shortest path between p and p', where $p_i \in \mathsf{Border}(I)$, and $p'_i \in \mathsf{Border}(I')$. By definition of the graph, the length of $path_1$ is less or equal to $\mathsf{diam}(\mathcal{G})$. By Definition 4.3, $\mathsf{dist}(\mathcal{G}, p_i, p'_i) \geq 1$. By construction, $\mathsf{dist}(\mathcal{G}, p, p_i), \mathsf{dist}(\mathcal{G}, p', p'_i) \geq \lceil \frac{\mathsf{diam}(\mathcal{G})}{2} \rceil$. This implies that the distance between p and p' through this path is greater or equal to $\mathsf{diam}(\mathcal{G}) + 1$. This contradicts that $path_1$ is the shortest one between p and p'. □

In the following, a detailed proof for Algorithm 4.2 is given. The same proof applies to Algorithm 4.3, with the difference that the $\mathsf{stab}_\mathcal{X}$ and $\mathsf{tail}_\mathcal{X}$ have different values. Therefore, for Algorithm 4.3, only a proof sketch showing the difference is provided.

Theorem 4.2. Algorithm 4.2 is self-stabilizing wrt. ME in $\lceil \mathsf{diam}(\mathcal{G})/2 \rceil - 1$ steps.

Proof. Let $\gamma_0, \gamma_1, \ldots$ be an execution. By Lemma 4.4, SU holds in $\gamma_{2\mathsf{diam}(\mathcal{G})}, \gamma_{2\mathsf{diam}(\mathcal{G})+1}, \ldots,$ which, by Lemma 4.5, implies that ME holds in $\gamma_{2\mathsf{diam}(\mathcal{G})}, \gamma_{2\mathsf{diam}(\mathcal{G})+1}, \ldots$. This implies that the liveness property of ME holds. It remains to show that for $\epsilon \leq i < 2 \cdot \mathsf{diam}(\mathcal{G})$, where $\epsilon = \lceil \frac{\mathsf{diam}(\mathcal{G})}{2} \rceil - 1$, the safety property of ME holds in γ_i; i.e. at most one process is privileged in γ_i.

Assume by contradiction that there exist two processes $p_a, p_b \in \mathcal{P}$, such that p_a and p_b are privileged in γ_i. Since

1. by definition of *privileged*: $\mathsf{r}_a^i \neq \mathsf{r}_b^i$, $\mathsf{r}_a^i, \mathsf{r}_b^i \geq \epsilon$ hold, and
2. by definition, $\mathsf{stab}_\mathcal{X} = \{0, \ldots, \mathsf{n}+\epsilon-1\}$ holds,

it follows by definition of the commands that r_a and r_b have values greater or equal to 0 in the configurations $\gamma_{i-\epsilon}, \ldots, \gamma_i$, and p_a and p_b execute only NA in each step of $(\gamma_{i-\epsilon}, \gamma_{i-\epsilon+1}), \ldots, (\gamma_{i-1}, \gamma_i)$. By definition of NA:

$\mathsf{r}_a^i = \varphi(\mathsf{r}_a^{i-1})$, and $\mathsf{r}_b^i = \varphi(\mathsf{r}_b^{i-1})$,
$\mathsf{r}_a^i = \varphi^2(\mathsf{r}_a^{i-2})$, and $\mathsf{r}_b^i = \varphi^2(\mathsf{r}_b^{i-2})$,
$\vdots$
$\mathsf{r}_a^i = \varphi^\epsilon(\mathsf{r}_a^{i-\epsilon})$, and $\mathsf{r}_b^i = \varphi^\epsilon(\mathsf{r}_b^{i-\epsilon})$.

Since $\mathsf{r}_a^i \neq \mathsf{r}_b^i$, it holds by the above analysis and by the definition of φ that $\mathsf{r}_a^{i-\epsilon} \neq \mathsf{r}_b^{i-\epsilon}$, which, by Definition 4.3, implies that p_a and p_b belong to two different islands in $\gamma_{i-\epsilon}$ (first deduction).

By definition of $\mathit{privileged}_{p_a}$, for each $q \in \mathcal{N}_{p_a}$, $\mathsf{r}_q^i = \mathsf{r}_a^i$ holds. The same holds for p_b. By Definition 4.4, this implies that each of the processes p_a, p_b belongs to an island of depth greater or equal to 1. Let $h_a \geq 1$ (resp. $h_b \geq 1$) be the depth of the island to which p_a (resp. p_b) belongs in γ_i. Since $\mathsf{r}_a^i, \mathsf{r}_b^i \geq \epsilon$, it follows by Lemma 4.6 that:

in γ_{i-1}, p_a (resp. p_b) belongs to a non-init-island of depth $h_a + 1$ (resp. $h_b + 1$),
in γ_{i-2}, p_a (resp. p_b) belongs to a non-init-island of depth $h_a + 2$ (resp. $h_b + 2$),
$\vdots$
in $\gamma_{i-\epsilon}$, p_a (resp. p_b) belongs to an island of depth $h_a + \epsilon$ (resp. $h_b + \epsilon$).

Since $h_a, h_b \geq 1$, and $\epsilon = \lceil \frac{\mathsf{diam}(\mathcal{G})}{2} \rceil - 1$ by construction, it follows that $(h_a + \epsilon), (h_b + \epsilon) \geq \lceil \frac{\mathsf{diam}(\mathcal{G})}{2} \rceil$. By Lemma 4.7, there exists at most one island I'', such that $\mathsf{idepth}(I'') \geq \lceil \frac{\mathsf{diam}(\mathcal{G})}{2} \rceil$. This implies that p_a and p_b belong to the same island in $\gamma_{i-\epsilon}$. This is a contradiction to the first deduction that p_a and p_b belong to two different islands. □

The following theorem concerns Algorithm 4.3 only.

Theorem 4.3. Algorithm 4.3 is self-stabilizing wrt. ME in $\lceil \mathsf{diam}(\mathcal{G})/2 \rceil - 1$ steps.

Proof Sketch. Similar to the proof argument for Theorem 4.2 with the following difference: the set $\mathsf{stab}_\mathcal{X}$ is equal to $\{-\epsilon, \ldots, \mathsf{n}-1\}$ instead of $\{0, \ldots, \mathsf{n}+\epsilon-1\}$, and $\mathsf{tail}_\mathcal{X}$ is equal to $\{-(\mathsf{diam}(\mathcal{G}) + \epsilon), \ldots, -\epsilon\}$ instead of $\{-\mathsf{diam}(\mathcal{G}), \ldots, 0\}$. □

4.4.3 Self-Stabilization wrt. $priv_\Delta$

In this section, it is shown that Algorithms 4.1–4.3 converge wrt. Δ recurrence of granting a unique privilege to any process p; i.e. Δ recurrence of *privileged* for any p, where Δ equals 1.0 for each of Algorithms 4.1 and 4.3, and $\mathsf{n}/(\mathsf{n} + \lceil \mathsf{diam}(\mathcal{G})/2 \rceil - 1)$ for Algorithm 4.2. The convergence time complexity wrt. Δ recurrence is also shown for each algorithm.

A condition *priv* is used to denote that there exists a process p, such that predicate *privileged* holds.

Definition 4.5 (*priv*)**.** A condition *priv* is said to be satisfied in a configuration γ of a topology $\mathcal{G} = (\mathcal{P}, \mathcal{E})$ iff there exists a process $p \in \mathcal{P}$ such that *privileged* holds in γ. $\diamondsuit$

First, the recurrence in Algorithms 4.1 and 4.3 is considered.

Lemma 4.8. For each configuration γ_0 of a topology $\mathcal{G} = (\mathcal{P}, \mathcal{E})$, if SU holds, and for each process $p \in \mathcal{P}$, $\mathsf{r}_p^0 \in \{0, \ldots, \mathcal{K}{-}1\}$, then for each execution $\gamma_0, \gamma_1, \ldots$ of any of Algorithms 4.1 and 4.3, the property $priv_{1.0}$ (i.e. $priv_\Delta$ where $\Delta = 1.0$) holds.

Proof. By Definition 3.2, $priv_{1.0}$ holds in $\gamma_0, \gamma_1, \ldots$ iff *priv* holds for each configuration $\gamma_0, \gamma_1, \ldots$. This is proven by induction:

- For the base case γ_0, by hypothesis, SU holds, and for each process p, $\mathsf{r}_p^i \in \{0, \ldots, \mathcal{K}{-}1\}$ holds. Given that each of $\{0, \ldots, \mathcal{K}{-}1\}$ belongs to a process id, by definition of *privileged*, one process is privileged in γ_i, implying that *priv* holds in γ_i.
- The inductive step: let γ_i be a configuration, in which SU and $\mathsf{r}_p^i \in \{0, \ldots, \mathcal{K}{-}1\}$ hold. It is shown that *priv* holds in the following configuration γ_{i+1}. By Lemma 4.1, SU holds in γ_{i+1}. By definition of the commands, only command NA is enabled, whose result keeps $\mathsf{r}_p^{i+1} \in \{0, \ldots, \mathcal{K}{-}1\}$ for each process p. This implies that *priv* holds in γ_{i+1}. $\square$

Theorem 4.4. Each Algorithm 4.1 and 4.3 is self-stabilizing wrt. $priv_{1.0}$ in $\alpha{+}\mathsf{diam}(\mathcal{G})$ steps.

Proof. Let γ_0 be a configuration. By Corollary 4.1, for each execution $\gamma_0, \gamma_1, \ldots$ of any of Algorithms 4.1 and 4.3, no process executes RA after $\gamma_{\mathsf{diam}(\mathcal{G})}$. Let p be a process, such that r_p is the minimum among all processes in $\gamma_{\mathsf{diam}(\mathcal{G})}$. If $\mathsf{r}_p \in \mathsf{stab}_\mathcal{X}$, then SU holds, because no process executes RA after $\gamma_{\mathsf{diam}(\mathcal{G})}$. If $\mathsf{r}_p \in \mathsf{tail}^*_\mathcal{X}$, in the step $(\gamma_{\mathsf{diam}(\mathcal{G})}, \gamma_{\mathsf{diam}(\mathcal{G})+1})$, $\mathsf{r}_p^{\mathsf{diam}(\mathcal{G})+1} = \mathsf{r}_p^{\mathsf{diam}(\mathcal{G})} + 1$, and since no process executes RA, then by definition of the commands, for each process $q \in \mathcal{P}$, $\mathsf{r}_q^{\mathsf{diam}(\mathcal{G})+1} \geq \mathsf{r}_p^{\mathsf{diam}(\mathcal{G})+1}$ holds. Inductively, in $\gamma_{\alpha+\mathsf{diam}(\mathcal{G})}$, it holds that $\mathsf{r}_q^{\alpha+\mathsf{diam}(\mathcal{G})} \geq 0$. By Lemma 4.4, each of Algorithms 4.1 and 4.3 is self-stabilizing wrt. SU in $2 \cdot \mathsf{diam}(\mathcal{G})$ steps. Since $\alpha \geq \mathsf{diam}(\mathcal{G})$, it follows that SU holds in $\gamma_{\alpha+\mathsf{diam}(\mathcal{G})}$. This, by Lemma 4.8, implies that $priv_{1.0}$ holds in $\gamma_{\alpha+\mathsf{diam}(\mathcal{G})}, \gamma_{\alpha+\mathsf{diam}(\mathcal{G})+1}, \ldots$. $\square$

Note that $\alpha = \mathsf{diam}(\mathcal{G})$ in Algorithm 4.1, and $\alpha = \lceil 1.5 \cdot \mathsf{diam}(\mathcal{G}) \rceil - 1$ in Algorithm 4.3. This implies that $priv_{1.0}$ holds in $2{\cdot}\mathsf{diam}(\mathcal{G})$ steps for Algorithm 4.1, and $\lceil 2.5{\cdot}\mathsf{diam}(\mathcal{G}) \rceil - 1$ steps for Algorithm 4.3.

The following lemma and the theorem concern Algorithm 4.2.

Lemma 4.9. Let $\Delta = \mathsf{n}/(\mathsf{n}+\epsilon)$. For each execution $\gamma_0, \gamma_1, \ldots$ of Algorithm 4.2 over a topology $\mathcal{T} = (\mathcal{P}, \mathcal{E})$, if for each process $p \in \mathcal{P}$, $\mathsf{r}_p^0 = \epsilon$ holds, then the condition $priv_\Delta$ holds.

Proof. Assume that $\forall p \in \mathcal{P} \bullet \mathsf{r}_p^0 = \epsilon$ holds in γ_0. By Definition 4.2, SU holds in γ_0, and by Lemma 4.1, SU holds in $\gamma_1, \gamma_2, \ldots$. Given that $\mathsf{r}_p^0 \in \mathsf{stab}_\mathcal{X}$, by definition of the commands, only the command NA is executed by all processes in the following steps. This implies that for all $1 \leq i \leq \mathsf{n}-1$, for all $p \in \mathcal{P}$, $\mathsf{r}_p^i = \varphi^i(\mathsf{r}_p^0)$. By definition of φ, it follows that $\mathsf{r}_p^i \in \{\epsilon, \ldots, \mathsf{n}{+}\epsilon{-}1\}$, and $\mathsf{r}_p^{\mathsf{n}-1} = \mathsf{n}{+}\epsilon{-}1$. By definition of $privileged_p$, in each configuration γ_i, there exists one process that is privileged, which implies that the number of configurations that satisfy *privileged* in $\gamma_0, \ldots, \gamma_{\mathsf{n}-1}$ is n. This implies that for each execution prefix $\gamma_0, \ldots, \gamma_i$, the recurrence of *priv* is 1.0, which is greater than Δ.

By definition of NA and by the above analysis, $\varphi^{\mathsf{n}+\epsilon}(\mathsf{r}_p^0) = \mathsf{r}_p^0$, and the recurrence of *privileged* in $\gamma_0, \ldots, \gamma_{\mathsf{n}+\epsilon-1}$ is greater or equal to Δ. Now the configuration $\gamma_{\mathsf{n}+\epsilon}$ is equal to γ_0. This, by definition of NA and φ, implies that the following execution suffix repeats the cycle $\gamma_0, \ldots, \gamma_{\mathsf{n}+\epsilon-1}$. Since this cycle always starts with n configurations satisfying *priv*, and is followed by only ϵ configurations, the recurrence of *priv* is greater or equals Δ in any $\gamma_0, \ldots, \gamma_j$ for $j \geq 0$. This implies that $priv_\Delta$ holds in $\gamma_0, \gamma_1, \ldots$. □

Theorem 4.5. Let $\Delta = \mathsf{n}/(\mathsf{n}{+}\epsilon)$. Algorithm 4.2 is self-stabilizing wrt. $priv_\Delta$ in $\max\{(\lceil 2.5 \cdot \mathsf{diam}(\mathcal{G})\rceil - 1), (\mathsf{n} + \lceil \mathsf{diam}(\mathcal{G})/2\rceil - 2)\}$ steps.

Proof. Let $\Delta = \mathsf{n}/(\mathsf{n}{+}\epsilon)$. By Lemma 4.9, any execution $\gamma_0, \gamma_1, \ldots$ of Algorithm 4.2 over a topology $\mathcal{G} = (\mathcal{P}, \mathcal{E})$ satisfies $priv_\Delta$ if for all $p \in \mathcal{P}$, $\mathsf{r}_p^0 = \epsilon$. To show the convergence time to reach a configuration, in which $\mathsf{r}_p = \epsilon$ for all $p \in \mathcal{P}$, two cases are distinguished:

1. SU holds in γ_0. In this case, for all processes p, q, $\mathsf{r}_p^0 = \mathsf{r}_q^0$, and in any following configuration, only NA or CA may be enabled for any process. If $\mathsf{r}_p^0 \in \{-\mathsf{diam}(\mathcal{G}), \ldots, \epsilon\}$, then in at most $\mathsf{diam}(\mathcal{G}) + \epsilon$ ($= \lceil 1.5 \cdot \mathsf{diam}(\mathcal{G})\rceil - 1$) steps, a configuration is reached where $\mathsf{r}_p = \epsilon$. Otherwise, if $\mathsf{r}_p^0 \in \{\epsilon + 1, \ldots, \mathsf{n}{+}\epsilon{-}1\}$, then a configuration, where $\mathsf{r}_p = \epsilon$ is reached in at most $\mathsf{n}{+}\epsilon{-}1$ ($= \mathsf{n} + \lceil \mathsf{diam}(\mathcal{G})/2\rceil - 2$) steps.

2. SU does not hold in γ_0. In this case, for any process p with the minimum value of r in γ_0, if $\mathsf{r}_p^0 \in \mathsf{stab}_\mathcal{X}$, then within $\mathsf{diam}(\mathcal{G})$ steps, at least one process executes RA. Otherwise, $\mathsf{r}_p^0 \in \mathsf{tail}_\mathcal{X}^*$. In both situations, by Lemma 4.2 and Corollary 4.1, after at most $\mathsf{diam}(\mathcal{G}) + \alpha = 2 \cdot \mathsf{diam}(\mathcal{G})$ steps, a configuration γ_j is reached, in which SU holds and for each process p, $\mathsf{r}_p^j = 0$. From γ_j, the processes execute NA, and after ϵ steps, $\mathsf{r}_p^{j+\epsilon} = \epsilon$ holds. This sums up to $\lceil 2.5 \cdot \mathsf{diam}(\mathcal{G})\rceil - 1$ steps.

Considering both cases, the convergence time to achieve $priv_\Delta$ is $\max\{(\lceil 2.5 \cdot \mathsf{diam}(\mathcal{G})\rceil - 1), (\mathsf{n} + \lceil \mathsf{diam}(\mathcal{G})/2\rceil - 2)\}$ □

Note that the convergence time complexity wrt. $priv_\Delta$ for each of the Algorithms is a lower bound. This can be shown by a straightforward construction of examples.

4.5 Convergence Time Optimality under the Synchronous Scheduler

The idea of the optimality proof of the ME-convergence time complexity under the synchronous scheduler was given by Dubois et al. [DG13]. It shows that the optimal convergence time wrt. ME is $\lceil \mathsf{diam}(\mathcal{G})/2 \rceil$. However, as observed in Section 4.3, there exist algorithms with an ME-convergence time of $\lceil \mathsf{diam}(\mathcal{G})/2 \rceil - 1$. In this section, the lower bound proof of the ME-convergence time complexity of [DG13] is refined; it is shown that $\lceil \mathsf{diam}(\mathcal{G})/2 \rceil - 1$ is the optimal one.

Two definitions and one Lemma are borrowed from [DG13].

Definition 4.6 (Local State [DG13])**.** Given a configuration γ, a process p and an integer $0 \leq k \leq \mathsf{diam}(\mathcal{G})$. The *k-local* state of p in γ (denoted by $\gamma_{p,k}$) is the configuration of the communication subgraph $\mathcal{G}' = (\mathcal{P}', \mathcal{E}')$ induced by $\mathcal{P}' = \{p' \in \mathcal{P} \mid \mathsf{dist}(\mathcal{G}, p, p') \leq k\}$ defined by $\forall\, p' \in \mathcal{P}' \bullet \gamma_{p,k}(p') = \gamma(p')$.[1] ♢

Definition 4.7 (Restrictions of an Execution [DG13])**.** Given an execution $\Xi = (\gamma_0, \gamma_1), (\gamma_1, \gamma_2), \ldots$ and a process p, the *restriction of Ξ to p* (denoted by Ξ_p) is defined by: $\Xi_p = (\gamma_0(p), \gamma_1(p)), (\gamma_1(p), \gamma_2(p)), \ldots$. ♢

Lemma 4.10 ([DG13])**.** Let γ, γ' be two configurations such that there exists a process p and an integer $1 \leq k \leq \mathsf{diam}(\mathcal{G})$ satisfying $\gamma_{p,k} = \gamma'_{p,k}$. Let A be a self-stabilizing algorithm wrt. ME for synchronous executions. The restrictions to p of the prefixes of length k of the synchronous executions of A starting respectively from γ and γ' are equal.

In the following, Theorem 4.6 is presented, which is a refined version of Theorem 4 of [DG13]. Next, the flaw of Theorem 4 of [DG13] is pointed out. Theorem 4.6 shows that $\lceil \mathsf{diam}(\mathcal{G})/2 \rceil - 1$ is the optimal ME-convergence time.

Theorem 4.6. The ME-convergence time of any self-stabilizing distributed algorithm wrt. ME is greater or equal to $\lceil \mathsf{diam}(\mathcal{G})/2 \rceil - 1$ (if $\mathsf{diam}(\mathcal{G}) > 0$) under the synchronous scheduler.

Proof. The claim holds trivially for $\mathsf{diam}(\mathcal{G}) = 1$. In the following, cases where $\mathsf{diam}(\mathcal{G}) \geq 2$ are considered. Let A be a self-stabilizing distributed algorithm wrt. ME, and let t be the ME-convergence time of A. Assume, by contradiction, that $t < \lceil \frac{\mathsf{diam}(\mathcal{G})}{2} \rceil - 1$ (Note that $\lceil \frac{\mathsf{diam}(\mathcal{G})}{2} \rceil - 1 = \lceil \frac{\mathsf{diam}(\mathcal{G}) - 2}{2} \rceil$).

Let $\mathcal{G} = (\mathcal{P}, \mathcal{E})$ be an arbitrary graph, and let p, q be two processes in $\mathcal{P}$ such that $\mathsf{dist}(\mathcal{G}, p, q) = \mathsf{diam}(\mathcal{G})$. Let $\Xi = (\gamma_0, \gamma_1), (\gamma_1, \gamma_2), \ldots$ be an execution starting from a configuration γ_0.

By the liveness property of ME, Ξ contains an infinite suffix in which p (resp. q) is privileged infinitely often. Hence, there exists a configuration γ_i (resp. γ_j) such that p (resp. q) is privileged in γ_i (resp. γ_j) and $i > t$ (resp. $j > t$).

By construction, since $\mathsf{dist}(\mathcal{G}, p, q) = \mathsf{diam}(\mathcal{G})$, and $\mathsf{diam}(\mathcal{G}) \geq 2$, it follows that there exists a path $p, p', \ldots, q', q$, such that $\mathsf{dist}(\mathcal{G}, p', q') = \mathsf{diam}(\mathcal{G}) - 2$ (p' and q' might be identical).

[1] By definition, $\gamma_{p,0} = \gamma(p)$.

Since $t < \lceil \frac{\mathsf{diam}(\mathcal{G})-2}{2} \rceil$, there exists at least one configuration γ'_0, such that $(\gamma'_0)_{p',t} = (\gamma_{i-t})_{p',t}$ and $(\gamma'_0)_{q',t} = (\gamma_{j-t})_{q',t}$. Let $\Xi' = (\gamma'_0, \gamma'_1), (\gamma'_1, \gamma'_2), \ldots$ be the synchronous execution of A starting from γ'_0.

By Lemma 4.10, one can deduce that the restriction to p' of the prefix of length t of Ξ' is the same as the one of the suffix of Ξ starting from γ_{i-t}. In addition, by definition of the graph, for all $g \in \mathcal{N}^*_p$, $\mathsf{dist}(\mathcal{G}, g, q) \geq \mathsf{dist}(\mathcal{G}, p', q)$, because the path $p, p', \ldots, q, q'$ is the shortest between p and q. This, analogously, implies that the restriction to g of the prefix of length t of Ξ' is the same as the one of the suffix of Ξ starting from γ_{i-t}.

By definition, a process's variables are visible only to the process itself and its neighbors. Hence, the privilege condition can be defined only over the variables of the process and all its neighbors. This, by construction of γ_i, implies that p is privileged in γ'_t. By the same deduction, q is also privileged in γ'_t. This contradiction leads to the result. □

In Theorem 4 of [DG13], the issue, that the privilege condition of a process p may also cover the states of the neighbors of p, is missed. In other words, the privilege condition is considered in [DG13] as if it covers only the local state of p. With this consideration, Theorem 4 of [DG13] concluded that the ME-convergence time is lower bounded by $\lceil \mathsf{diam}(\mathcal{G})/2 \rceil$. This consideration is not necessarily required in the shared memory model, formalized in Chapter 2.

4.6 Remarks

The three algorithms exploit a trade-off between many aspects of the performance of self-stabilizing mutex algorithms: the convergence wrt. mutex, the Δ recurrence that can be achieved, the convergence wrt. Δ recurrence, and the space requirement. Table 4.1 summarizes the time and space complexities of Algorithms 4.1–4.3. The space complexity represents the size of the local state space for each process.

	Algorithm 4.1	Algorithm 4.2	Algorithm 4.3
ME-Convergence Time	$\mathsf{diam}(\mathcal{G})-1$	$\lceil \mathsf{diam}(\mathcal{G})/2 \rceil -1$	$\lceil \mathsf{diam}(\mathcal{G})/2 \rceil -1$
Recurrence Δ	1.0	$\mathsf{n}/(\mathsf{n} + \lceil \mathsf{diam}(\mathcal{G})/2 \rceil -1)$	1.0
$priv_\Delta$-Convergence Time	$2 \cdot \mathsf{diam}(\mathcal{G})$	$\max\{(\lceil 2.5 \cdot \mathsf{diam}(\mathcal{G}) \rceil - 1),$ $(\mathsf{n} + \lceil \mathsf{diam}(\mathcal{G})/2 \rceil - 2)\}$	$\lceil 2.5 \cdot \mathsf{diam}(\mathcal{G}) \rceil - 1$
Space	$\mathsf{n} + \mathsf{diam}(\mathcal{G})$	$\mathsf{n} + \lceil 1.5 \cdot \mathsf{diam}(\mathcal{G}) \rceil - 1$	$\mathsf{n} + \lceil 1.5 \cdot \mathsf{diam}(\mathcal{G}) \rceil - 1$

Table 4.1: Time and space complexity for Algorithms 4.1–4.3

As observed, reducing the convergence time wrt. mutex or the space requirement – which are the typical performance aspects that are usually considered – may affect another aspect of the performance: a fast convergence wrt. mutex does not imply a fast installation of the desired recurrence. This trade-off is obvious concerning Algorithms 4.1 and 4.2, where both are based on the finite incrementing system presented in Section 4.3.1. On one hand, Algorithm 4.1 guarantees that in $2 \cdot \mathsf{diam}(\mathcal{G})$ steps, both ME and $priv_{1.0}$ are achieved, while Algorithm 4.2 does not. On the other hand, Algorithm 4.2 converges to ME faster than Algorithm 4.1 does. Algorithm 4.3

is an enhanced version of Algorithm 4.2 (by re-engineering the incrementing system of [BPV08]): it can still achieve 1.0 recurrence of *priv*. However, it may not converge wrt. $priv_{1.0}$ in $2 \cdot \mathsf{diam}(\mathcal{G})$ steps as Algorithm 4.1 guarantees.

The basic design of the finite incrementing system considered solving self-stabilization wrt. mutex in the light of the synchronous unison property [BPV08]. By considering this design, the authors followed an intuitive reset phase – represented by $\mathsf{tail}^*_{\mathcal{X}}$ – and a live phase $\mathsf{stab}_{\mathcal{X}}$ in which the system is supposed to keep incrementing the values. By looking into the design of Algorithm 4.3 regardless of the recurrence properties, it does not make sense to have a subset of $\mathsf{stab}_{\mathcal{X}}$, which is $\{-\epsilon, \ldots, -1\}$, outside the incrementing phase after the system converges. The significance of such a subset appeared in the light of the recurrence properties.

Recurrence can be extended to evaluate other performance properties in relation to mutex. For example, in the local mutex and group mutex problems, recurrence can be extended to capture the number of privileged processes in one configuration. In other words, the ratio Δ can be redefined to capture more details about each configuration rather than counting configurations satisfying some condition. This can be achieved by considering the ratio of local states that satisfy a condition in each configuration, in the calculation of Δ.

5 Fast and Educated Unique Process Selection

This chapter focuses on a particular problem that is correlated to mutual exclusion. The problem is called *unique process selection*. The problem concerns granting a unique privilege to processes to access the critical section, however, with a special consideration of the fairness property. With respect to fairness, a basic and an extended case are considered. In the basic case, the fairness property is neglected in the design of the algorithm, such that fairness is still satisfied if the algorithm is highly recurrent in selecting processes to be granted privilege. In the extended case, the choice of processes to be granted a privilege are based on local or global criterion, which is referred to as *educated unique process selection*. The notion of Propagation of Information with Feedback (PIF) [Cha82, Seg83] is exploited to design self-stabilizing algorithms wrt. unique process selection for each case. The algorithms are designed for tree topologies, since exactly one process – namely the root – is required to do the selection. The aim and challenge is to achieve a high recurrence of granting a privilege, given particular environments. Some of this chapter's content has appeared in [4, 5] of the author's publications.

The structure of this chapter is as follows. Section 5.1 introduces and motivates the unique process selection problem. Section 5.2 presents the problem statement and states the contributions. Next, Section 5.3 explains the PIF approach and its related work. Section 5.4 presents an algorithm for fast unique process selection. Next, Section 5.5 presents an algorithm for educated unique process selection. Section 5.6 provides correctness proofs. Finally, Section 5.7 presents a discussion.

5.1 The Unique Process Selection Problem

The fairness property in mutual exclusion ensures that each process is granted a privilege to access the critical section infinitely often (cf. Section 4.1). The intuition of fairness is to ensure that each process gets its chance to execute its action infinitely often, without waiting forever. In general, without fairness, some processes may be blocked forever, which may result in critical consequences.

To satisfy the fairness property by an algorithm, the algorithm's design may be impacted. For example, token-passing approaches require that the token is circulated in some manner, such that it reaches all processes in each cycle. In the design of self-stabilizing mutual exclusion algorithms, there is usually an implicit assumption that each process continuously requests a privilege to access the critical section (cf. Section 4.1.2). Due to this assumption, the impact of satisfying the fairness requirement is seen to be positive from all aspects. However, if the number of processes requesting a privilege is low, then the fairness impact may be negative regarding the algorithm's

performance; the process selection mechanism for granting a privilege may be low recurrent if only few processes are request a privilege. Instead, searching for and selecting requesting processes may be more efficient; the value of Δ is then a factor of the topology diameter or depth. Note that the issue of requesting a privilege or not by the processes is not considered explicitly in the area of self-stabilization. A motivational example of this follows.

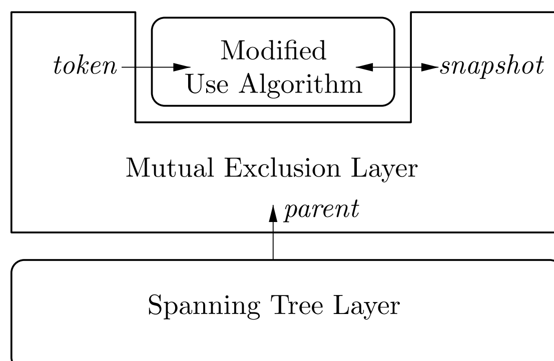
The example concerns a wireless sensor network consisting of sensors that are all linked to a central unit, forming a tree topology. The network is used to monitor environmental actions that rarely happen, e.g. fire or earthquakes. When a sensor detects an environmental action, then it is supposed to send an alarm message that is forwarded through a path to the central unit. In this network, the sensors are required to aggregate information fast given limited resources, such as a limited bandwidth or a limited message size. This requires that each sensor performs its task separately. In such a case, having fast selection of the sensors that are sending alarm messages to process or deliver the information is more useful than having a slow selection of all sensors for the sake of being fair. This trade-off is analyzed in detail in Section 5.7.

From this perspective, the problem of unique process selection is introduced. The problem simply concerns granting a unique privilege to processes, where it is guaranteed that granting a privilege happens infinitely often, but not necessarily for all processes. If an algorithm satisfies unique process selection, and the recurrence of granting a privilege is high enough, then the algorithm is useful in the environments, where processes rarely request a privilege. In other words: if the achieved recurrence is higher than the frequency of requesting a privilege by processes, then the classical fairness property is satisfied. For example, if recurrence of selecting a process is equal to 0.5, and at most one process requests a privilege every 10 steps, then fairness is satisfied anyway.

5.1.1 Educated Unique Process Selection

An extended version of unique process selection – presented in this section – addresses some particular environments, in which the actions are prioritized to guarantee better quality of service. An example of such an environment follows.

Consider the system given in [DT10]: it is a fair composition of three self-stabilizing algorithms, where one of them is a mutual exclusion algorithm and another one is referred to as a 'use algorithm' as follows:



In this composition, if the use algorithm is in an illegitimate configuration, the convergence (stabilization) of the algorithm can be observed by the mutual exclusion algorithm using a ranking function. Ranking functions are used to rank configurations, such that the decrease of the ranking function value in any execution indicates a convergence wrt. a legitimate configuration (e.g., [The00b, AKM+07]). In the approach [DT10], the mutual exclusion algorithm's task is to circulate a token that reaches all processes infinitely often, and to enable processes to execute actions of the use algorithm only if the ranking function's value decreases. The mutual exclusion algorithm, however, does not enforce priorities over the actions executed by the processes; it only guarantees fair execution. In such a case, it is helpful if the mutual exclusion algorithm would employ educated selection, based on which process is most promising to achieve fast convergence of the use algorithm, which can be known by observing the decrease of the ranking function value. Intuitively, during the convergence of the use algorithm, satisfying the fairness property is not as important as reducing the convergence time of the use algorithm, given that the safety property of mutual exclusion may anyway not be satisfied during convergence. Moreover, the selection can additionally be realized according to quality of service indicators after the system has stabilized.

From this point, the *educated unique process selection* problem is introduced. This problem concerns selecting processes to be granted a privilege, such that if a process is granted a privilege, then the process is distinguished from the others by some criterion. The criterion is modelled by a real number variable owned by each process, and a process is selected if it has the maximum value among all processes.

Since the criterion variable is owned by each process, the variable presents local criterion. To have process selection based on global criterion, this variable is updated according to a snapshot of the given configuration. A particular procedure of passing the snapshot to all processes is given in Section 5.5.

Note that educated selection collects some aspects from the mutual exclusion and the consensus problems [FLP85, DKS10]. However, it does not necessarily fulfill any of them. Moreover, educated selection is not intended to solve the leader election problem [Awe87, DIM97]; it aims to select potentially different processes frequently. In addition, educated selection can still be controlled to preserve the classical fairness property of mutual exclusion.

5.2 Problem Statement and Assumptions

The aim is to design two algorithms satisfying educated, and unique process selection, respectively, and achieving high recurrence of granting a privilege to processes. The specification of unique process selection is twofold. First, in each configuration, at most one process is privileged to access the critical section. Second, granting a privilege to an arbitrary process holds infinitely often, but not necessarily for all processes. This specification is formalized in the following definition.

Definition 5.1 (Unique Process Selection – *UPS*)**.** Let $\mathcal{G} = (\mathcal{P}, \mathcal{E})$ be a topology, let A be an algorithm, and let *privileged* be a condition on each state of each process p (or shortly $privileged_p$). A specification *UPS*, denoting **unique process selection**, is said to be satisfied by an execution $\Xi : \gamma_0, \gamma_1, \ldots$ of A over $\mathcal{G}$ iff:

1. Safety: if *privileged* holds for a process $p \in \mathcal{P}$ in a configuration γ_i, then *privileged* does not hold for any process $q \in \mathcal{P} \setminus \{p\}$ in γ_i.
2. Liveness: the condition *privileged* holds infinitely often, regardless of which process.[1] ♢

Educated unique process selection differs from the classical one by selecting processes only if they are distinguished by some criterion, abstracted by a real number variable.

Definition 5.2 (Educated Unique Process Selection – *EUPS*)**.** Let $\mathcal{G} = (\mathcal{P}, \mathcal{E})$ be a topology. Let A be an algorithm. Let *privileged* be a condition on each state of each process p, and let $v \in \mathbb{R}$ be a particular variable owned by each process. A specification *EUPS* wrt. v, denoting **educated unique process selection wrt.** v, is said to be satisfied by an execution $\Xi : \gamma_0, \gamma_1, \ldots$ of A over $\mathcal{G}$ iff:

1. Unique process selection is satisfied by Ξ.
2. If *privileged* holds for a process $p \in \mathcal{P}$ in a configuration γ_i, then for all $q \in \mathcal{P}$, $p.v \geq q.v$ holds. ♢

For simplicity, from now on, the expression "wrt. v" is omitted.

5.2.1 Problem Statement

The aim is to design two self-stabilizing algorithms that satisfy the introduced properties, where the algorithms show more efficiency than the typical token passing or mutex algorithms in terms of recurrence properties, for specific environments. The two algorithms are designed for tree topologies using the shared memory model. The choice of the scheduler is customized based on the algorithm. The problem statement formalization follows. Note that the efficiency analysis is given in Section 5.7.

Problem 5.1. Devise two self-stabilizing distributed algorithms for tree topologies using the shared memory model, such that both achieve high recurrent unique process selection, and the second provides an educated one.

The contribution of this chapter is as follows:

- An algorithm for unique process selection for tree topologies under the synchronous scheduler is presented. Let $\mathcal{T} = (\mathcal{P}, \mathcal{E})$ be a tree topology. The convergence time wrt. *UPS* is $\mathsf{depth}(\mathcal{T})$. The recurrence of granting a privilege is $\frac{1}{4 \cdot \mathsf{depth}(\mathcal{T})}$, and is achieved in $7 \cdot \mathsf{depth}(\mathcal{T})$ steps. In the average case, the algorithm achieves $\frac{1}{4 \cdot \mathsf{avg_depth}(\mathcal{T}))}$ recurrence of granting a privilege, where $\mathsf{avg_depth}(\mathcal{T}))$ is the average depth for any process in $\mathcal{P}$.
- An algorithm for educated unique process selection for tree topologies under the asynchronous scheduler is presented. The algorithm provides the same worst case time complexity and recurrence as the previous algorithm, if applied under the synchronous scheduler.

[1] In contrast to mutual exclusion, the condition *privileged* is not required to hold for each process infinitely often.

- Next, it is shown how the second algorithm is extended for educated selection based on a global criterion.

5.2.2 Assumptions

The assumptions in this chapter are as follows:

- A *topology* is a tree $\mathcal{T} = (\mathcal{P}, \mathcal{E})$, where $\mathcal{P}$ is a set of *processes*. Each process other than the root is called a *non-root* process.
- The *communication model* used by the topologies is the shared memory model.
- The number of processes n is known by each process.

5.3 PIF: Propagation of Information with Feedback

Propagation of Information with Feedback – shortly *PIF* – is a sort of wave or echo distributed algorithm [Cha82, Seg83, KRS84]. In a PIF approach, a process starts a so-called PIF cycle by sending a message to its neighbors. Each neighbor forwards the message to the other neighbors, until the message reaches all processes. Next, an acknowledgement – or a feedback – is sent from each process. Feedbacks sent from all processes are received by original message sender, which terminates the PIF cycle.

PIF is useful for many tasks in distributed systems, like snapshot maintenance, infimum or supremum computation, and synchronization. Examples are found in [RH90, Lyn96, Tel00]. In particular, the nature of communication in PIF makes it more useful for collecting and aggregating information by one process than token ring approaches [HMR94] and the approaches presented in Section 4.3; PIF exploits the topology diameter or depth by sending multiple tokens to collect information. In this work, PIF is chosen to be applied for designing self-stabilizing algorithms satisfying educated and unique process selection. The advantage of PIF over other approaches concerning unique process selection is illustrated in detail in Section 5.7.

The specification of PIF indicates that when a message is sent from an initiating process, after a finite number of steps, all processes acknowledge the reception of the message to the initiating process. In the scope of this work, PIF is used as a token-passing methodology to satisfy educated and unique process selection. It is, however, not mandatory to satisfy all specifics of PIF. Therefore, the specification of PIF does not appear in the formal analysis of the algorithms, and is not necessarily fulfilled in all cases. Instead, the properties educated and unique process selection are analyzed.

5.3.1 Related Work

Early PIF approaches appear in [Cha82, Seg83, KRS84], which present basic designs of PIF for many environments. Many following examples are given in [RH90, Lyn96, Tel00]. These solutions are not necessarily self-stabilizing with respect to PIF.

Self-stabilization wrt. PIF is tackled in an extensive work by Villain et al., e.g. [BDPV99c, BDPV99a, CPVD01, CDPV02, CDPV05, BDPV07, LMV14]. The work involves optimal solutions wrt. space requirement for trees [BDPV99a, CDPV05,

BDPV07] and for rooted networks where trees are built dynamically [CPVD01]: the space requirement is 2 values for each of the root and leaves, and 3 values for inner processes. The work of Villain also considers many topological aspects: trees with unknown sense of direction [BDPV99c], unoriented trees [CDPV05, BDPV07],[1] arbitrary rooted networks [CPVD01], and arbitrary non-rooted networks in which process ids are exploited to form a tree [CDPV02]. A recent work considers PIF using the message passing model [LMV14]. Concerning the convergence time wrt. PIF, the algorithms given in [BDPV99d, BDPV99b, CDPV02, CDPV05, CDV06, BDPV07, LMV14] are snap-stabilizing; i.e. they stabilize to a correct behavior in 0 steps (cf. Chapter 1), where the correct behavior adheres to PIF.

Kruijer presents a self-stabilizing algorithm for trees in [Kru79]. The algorithm extends the 4-state-machine algorithm of Dijkstra [Dij74]. The algorithm allows multiple tokens to be sent from the root through the tree paths, such that there is one token per path. It is well known that this algorithm can be applied to trees as a PIF algorithm. In this work, a similar approach to [Kru79] is exploited.

5.3.2 Extending Dijkstra's 4-State-Machine Algorithm

The basic design concept in this work reflects searching for a process requesting a privilege, and selecting one, guaranteeing unique process selection. From now on, a process that is requesting a privilege is called an *active process*.

The basic scenario for searching for and selecting active processes uniquely is achieved following a PIF-like approach over trees as follows (cf. Figure 5.1 where p_3 is active):

1. The root propagates a token that reaches all processes. The token's aim is to check whether there is an active process. The token is called *search token*.

2. After the token reaches the leaves, a feedback is sent to the root. The feedback informs the root about active processes. The token is called *feedback token*.

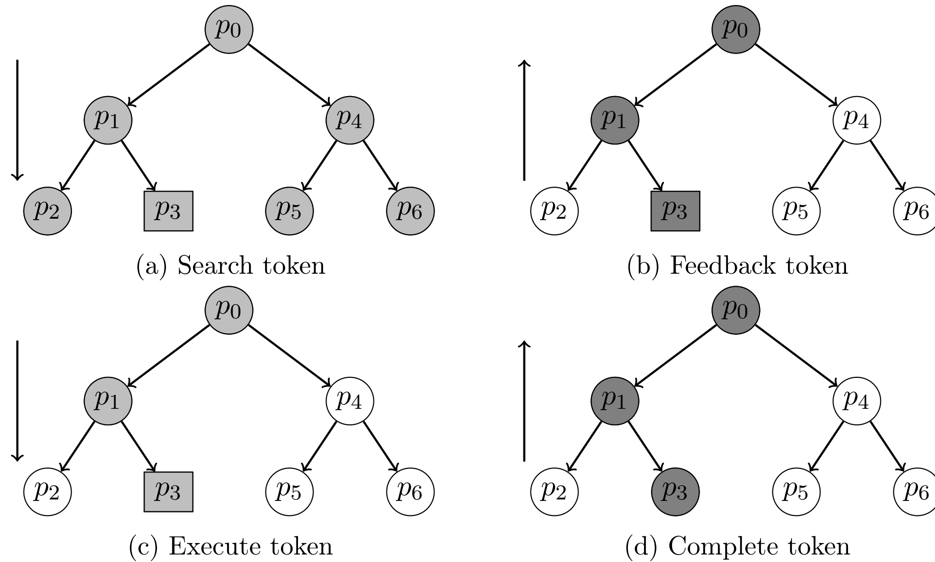
3. The root selects an active process – if exists – and targets a token to the active process. The token is called *execute token*.

4. The selected process executes its action, and sends back a token that notifies the root about the completion of the execution. The token is called *complete token*.

To achieve this scenario, the approach used is similar to Kruijer's [Kru79], which extends Dijkstra's 4-state-machine algorithm [Dij74]. This approach is chosen in particular for the following reasons:

- The 4-state-machine algorithm requires four values for each inner process, and two for each of the root and leaves, for the following reason: the four values are required to conveniently reflect the search, feedback, execute, and complete tokens, respectively,

[1] Unoriented trees are directed trees, where instead of one edge, a pair of symmetric edges between any parent and its child may exist.



(a) Search token (b) Feedback token (c) Execute token (d) Complete token

Figure 5.1: Desired scenario. The square denotes an active process

- The convergence time complexity to guarantee starting to search for an active process correctly, is $\mathcal{O}(\mathsf{depth}(\mathcal{T}))$, which is reasonably good. Note that some algorithms in the related work are snap-stabilizing wrt. PIF. However, if they are used to achieve the scenario of this work, they converge in $\mathcal{O}(\mathsf{depth}(\mathcal{T}))$ steps to achieve unique process selection.
- In the case of unique process selection, the chosen methodology realizes the synchronous scheduler to obtain immediate feedbacks, which increases the recurrence of granting a privilege on average.

Note that the followed approach is similar to the second one in [BDPV07] in the mechanism of sending waves back and forth between root and leaves. However, it differs in having four values per process instead of three.

Before illustrating the approach, the 4-state-machine algorithm is explained. This algorithm is designed to be executed over ring topologies. The algorithm behaves as follows: there are two special neighboring processes p and q in a ring, and a single token is sent back and forth between p and q through all the other processes. With this scenario, each process in the ring receives the token uniquely and infinitely often. More details about Dijkstra's algorithm and its complexity are found in [Dij74, Kru79, Dij86].

In this work, the scenario of Dijkstra's algorithm for ring topologies is exploited in trees by sending multiple tokens back and forth between the root and the leaves.

The basic extension of Dijkstra's algorithm to trees is presented in Algorithm 5.4 (recall that $\mathsf{Ch}(p)$ denote the set of children of a process p.) Each process has two boolean variables x, up, and the given guarded commands. It is asserted that always $\mathsf{up} = \top$ for the root, and $\mathsf{up} = \bot$ for each leaf, which implies that the root and each leaf has only two values.

Algorithm 5.4 works under the asynchronous scheduler, but it does not provide immediate feedbacks. The tokens propagated by the root always reach the leaves, and

Algorithm 5.4 Extended 4-state-machine algorithm to trees

// This represents the sub-algorithm for process p

Variables

$\mathsf{x} \in \mathbb{B}$, $\mathsf{up} \in \mathbb{B}$

Assertions

$\mathsf{root.up} = \top \;\wedge\; \forall p \in \mathsf{Leaves} \bullet p.\mathsf{up} = \bot$

Guarded Commands (gc_i : *guard* $\longrightarrow$ *action*)

<u>For Root</u>

$\mathsf{gc}_1 : \forall\, ch \in \mathsf{Ch}(p) \bullet ch.\mathsf{x} = \mathsf{x} \wedge \neg ch.\mathsf{up} \longrightarrow \mathsf{x} := \neg\mathsf{x}$

<u>For Leaves</u>

$\mathsf{gc}_2 : \mathsf{parent}(p).\mathsf{x} \neq \mathsf{x} \longrightarrow \mathsf{x} := \neg\mathsf{x};$

<u>For Inner Processes</u>

$\mathsf{gc}_3 : \mathsf{parent}(p).\mathsf{x} \neq \mathsf{x} \longrightarrow \mathsf{x} := \neg\mathsf{x};\ \mathsf{up} := \top;$

$\mathsf{gc}_4 : \neg\mathsf{up} \wedge \forall\, ch \in \mathsf{Ch}(p) \bullet ch.\mathsf{x} = \mathsf{x} \wedge \neg ch.\mathsf{up} \longrightarrow \mathsf{up} := \bot;$

feedback is given from all processes. Synchronicity is exploited for sending immediate feedbacks in Section 5.4.

Recall that the aim is to have four tokens by the four values. The guards of the commands of Algorithm 5.4 are represented as tokens in the following:

$\mathsf{token}_1 : \mathsf{parent}(p).\mathsf{x} \neq \mathsf{x} \wedge \neg\mathsf{x}$
$\mathsf{token}_2 : \mathsf{up} \wedge \mathsf{x} \wedge \forall\, ch \in \mathsf{Ch}(p) \bullet ch.\mathsf{x} = \mathsf{x} \wedge \neg ch.\mathsf{up}$
$\mathsf{token}_3 : \mathsf{parent}(p).\mathsf{x} \neq \mathsf{x} \wedge \mathsf{x}$
$\mathsf{token}_4 : \mathsf{up} \wedge \neg\mathsf{x} \wedge \forall\, ch \in \mathsf{Ch}(p) \bullet ch.\mathsf{x} = \mathsf{x} \wedge \neg ch.\mathsf{up}$

Following the above commands, token_1 and token_3 (resp. token_2 and token_4) are passed from parents to children (resp. children to parents). Every two tokens passed in the same direction are distinguished by the value of x. Algorithm 5.4 can be re-constructed using the above token notation as follows:

<u>For Root</u>

$\mathsf{gc}_1 : \mathsf{token}_2 \vee \mathsf{token}_4 \longrightarrow \mathsf{x} := \neg\mathsf{x}$

<u>For Leaves</u>

$\mathsf{gc}_2 : \mathsf{token}_1 \vee \mathsf{token}_3 \longrightarrow \mathsf{x} := \neg\mathsf{x};$

<u>For Inner Processes</u>

$\mathsf{gc}_3 : \mathsf{token}_1 \vee \mathsf{token}_3 \longrightarrow \mathsf{x} := \neg\mathsf{x};\ \mathsf{up} := \top;$

$\mathsf{gc}_4 : \mathsf{token}_2 \vee \mathsf{token}_4 \longrightarrow \mathsf{up} := \bot;$

Following the above notation, it is obvious that the actions for token_1 and token_3 are the same. This also holds for token_2 and token_4. In the following algorithms in Sections 5.4 and 5.5, the tokens $1, 2, 3$ and 4 are utilized, to reflect the search, feedback, execute, and complete tokens, respectively, where processes receiving different tokens execute different actions to achieve the aim.

5.4 Exploiting Synchronicity for Immediate Feedback

A self-stabilizing algorithm wrt. unique process selection is introduced in Algorithm 5.5. The idea behind Algorithm 5.5 is to have an immediate feedback, once an active process is found, without bothering about forwarding the token to the process's descendants. This can be achieved by exploiting the synchronous scheduler. Note that having immediate feedbacks may violate the specification of PIF, that each process should receive the token and send a feedback.

Each process owns the following variables:

- The variables $\mathsf{up} \in \mathbb{B}$ and $\mathsf{x} \in \mathbb{B}$ are taken from Algorithm 5.4. Similarly, it is asserted that always $\mathsf{up} = \top$ for the root, and $\mathsf{up} = \bot$ for each leaf.
- A new variable $\ell \in \{-1, 0, \ldots, \mathsf{n}-1\}$ is added. This variable is used to store process id's to mark active processes, and paths leading to active processes, in order to target some tokens. The valuation $\ell = -1$ denotes that ℓ is not pointing to any process.
- The flag *active* is added to denote whether a process is active. The value of *active* may change in any step independent of the algorithm; i.e. by an external action.

The function *critSection*() denotes the particular action – accessing the critical section – executed when a process is privileged.

The algorithm has tokens 1–4 reflecting the search, feedback, execute, and complete tokens, respectively. However, token_2, which is a feedback token has two sorts: positive and negative feedbacks. Informally, in the former case, the token informs about an active process in the subtree rooted by the process that is sending the token, while in the latter, the token informs that there is no active process in the corresponding subtree.

Note that each of the root, the inner processes, and the leaves have different sub-algorithms (cf. Algorithm 5.5). The stable behavior of Algorithm 5.5 is an infinite repetition of two cycles, where in each cycle, the root propagates a token, and receives at least one.

Informally, in the first cycle the root propagates a search token (token_1) searching for active processes. Once an active process receives the token, it immediately sends a positive feedback token ($\mathsf{token}_{2\text{-}a}$) to the root, and sets ℓ to the id of itself. When an ancestor receives a positive feedback token, it sets ℓ to the id of the process that is sending the token. If there is no active process in the topology, negative feedback tokens ($\mathsf{token}_{2\text{-}b}$) are sent from the leaves up to the root. In the second cycle, if the root receives a positive feedback token, then the root points to the child *ch* who is sending the token by setting ℓ to $ch.\mathsf{id}$, and the root sends an execute token (token_3). If the feedback is negative, the root sets $\ell = -1$. In the former case, the execute token is forwarded to an active process. The followed path is known from the values of ℓ. The active process – now the selected process – executes *critSection*(), and then, it sends a complete token (token_4) that is forwarded to the root. The root may execute *critSection*() after receiving the complete token.

To explain Algorithm 5.5 in detail, for convenience of presentation, some notation concerning the states of processes is introduced. Given a process p:

- The variables $p.\mathsf{x}$, $p.\mathsf{up}$, and $p.\ell$ are represented together as $p_{\mathsf{x}}^{\mathsf{up}}\ell$. For example, the notation $p_{\bot}^{\top}5$ denotes that $p.\mathsf{x} = \bot$, $p.\mathsf{up} = \top$, and $p.\ell = 5$.
- If only one value is specified, the sign "?" is sometimes written at the position of the other values. For example, $p_{\top}^{?}?$ or $p_{\top}^{?}$ denote that $p.\mathsf{x} = \top$ regardless of the values of up and ℓ.
- To denote that $p.\ell \neq -1$, given (for example) that $p.\mathsf{x} = a$ and $p.\mathsf{up} = b$, the notation $p_{a}^{b}\not{1}$ is used.

The stable execution of Algorithm 5.5 is an infinite repetition of two cycles, which are explained below in detail by referring to Algorithm 5.5.

First Cycle

$\mathsf{token}_1 \downarrow$: the root sends token_1 to its children. If a process p receives token_1, then there are two possible reactions:

- If p is active, then p switches into $p_{\top}^{\bot}p.\mathsf{id}$, sending to its parent $\mathsf{token}_{2\text{-}a}$ immediately (commands $\mathsf{gc}_6, \mathsf{gc}_{14}$).
- If p is not active, then p switches into $p_{\top}^{\top}$-1 to send token_1 to its children (gc_5) if p is an inner process, or it switches into $p_{\top}^{\bot}$-1 (gc_{13}) if p is a leaf, since for all leaves, $\mathsf{up} = \bot$.

$\mathsf{token}_{2\text{-}a} \uparrow$: If a process p receives $\mathsf{token}_{2\text{-}a}$ from a child q, then there exists an active process in the subtree rooted by p. p switches into $p_{\top}^{\bot}q.\mathsf{id}$ (gc_7) – The process p points to q. In the next steps, the ancestors of p pass $\mathsf{token}_{2\text{-}a}$ analogously as p did, where each parent points to its sending child, until $\mathsf{token}_{2\text{-}a}$ reaches the root.

$\mathsf{token}_{2\text{-}b} \uparrow$: If a process p receives $\mathsf{token}_{2\text{-}b}$, then there exists no active process in the subtree rooted by p. p switches into $p_{\top}^{\bot}$-1 (gc_8). If there is no active process in the whole topology, each leaf switches into $p_{\top}^{\bot}$-1 (gc_{13}) after receiving token_1, and it follows that each parent of a leaf receives $\mathsf{token}_{2\text{-}b}$ from its children and forwards it upwards (gc_8) until it reaches the root.

Second Cycle

$\mathsf{token}_3 \downarrow$: The root receives $\mathsf{token}_{2\text{-}a}$ if there is at least one active non-root process in the tree or $\mathsf{token}_{2\text{-}b}$ otherwise. The root sends token_3 (gc_1, gc_2), such that if there is an active process in the tree, the value of $\mathsf{root}.\ell$ is set to the id of the child that is linked to an active process, whose feedback was received by the root at earliest (gc_1). If there is no active process, then the value of $\mathsf{root}.\ell$ is set to -1 (gc_2). If a process p receives token_3, one of three cases may exist:

1. Case (1) represented by $\mathsf{gc}_{11}, \mathsf{gc}_{16}$: if $\mathsf{parent}(p)$ is not pointing to p; i.e. $\mathsf{parent}(p).\ell \neq p.\mathsf{id}$, or p is neither pointing to itself nor to one of its children, this implies that there is no selected process in the subtree rooted by p. The process p, then, switches into $p_{\bot}^{\bot}$, where p does not point to any of its children. Note that the children of p (if they exist) behave the same in the next step. If p is a leaf, then it does not need to check if it is pointing to itself since it has no children. In this case, no process in the subtree rooted by p gets a privilege in the current cycle.

Algorithm 5.5 Unique process selection by exploiting synchronicity for immediate feedback in PIF

// This represents the sub-algorithm for process p

Variables

$\mathsf{x} \in \mathbb{B}$
$\mathsf{up} \in \mathbb{B}$
$\ell \in \{-1, 0, \ldots, \mathsf{n}-1\}$
$active \in \mathbb{B}$

Assertions

$\mathsf{root.up} = \top \ \wedge\ \forall q \in \mathsf{Leaves} \bullet q.\mathsf{up} = \bot$

Tokens

$\mathsf{token}_1 : \mathsf{parent}(p).\mathsf{x} \neq \mathsf{x} \wedge \neg\mathsf{x}$ % Search Token
$\mathsf{token}_{2\text{-}a} : \mathsf{up} \wedge \mathsf{x} \wedge \exists\, ch \in \mathsf{Ch}(p) \bullet ch.\mathsf{x} = \mathsf{x} \wedge \neg ch.\mathsf{up} \wedge ch.\ell \neq -1$ % Positive Feedback Token
$\mathsf{token}_{2\text{-}b} : \mathsf{up} \wedge \mathsf{x} \wedge \forall\, ch \in \mathsf{Ch}(p) \bullet ch.\mathsf{x} = \mathsf{x} \wedge \neg ch.\mathsf{up} \wedge ch.\ell = -1$ % Negative Feedback Token
$\mathsf{token}_3 : \mathsf{parent}(p).\mathsf{x} \neq \mathsf{x} \wedge \mathsf{x}$ % Execute Token
$\mathsf{token}_4 : \mathsf{up} \wedge \neg\mathsf{x} \wedge \forall\, ch \in \mathsf{Ch}(p) \bullet ch.\mathsf{x} = \mathsf{x} \wedge \neg ch.\mathsf{up}$ % Complete Token

Guarded Commands (gc_i : *guard* $\longrightarrow$ *action*)

For Root

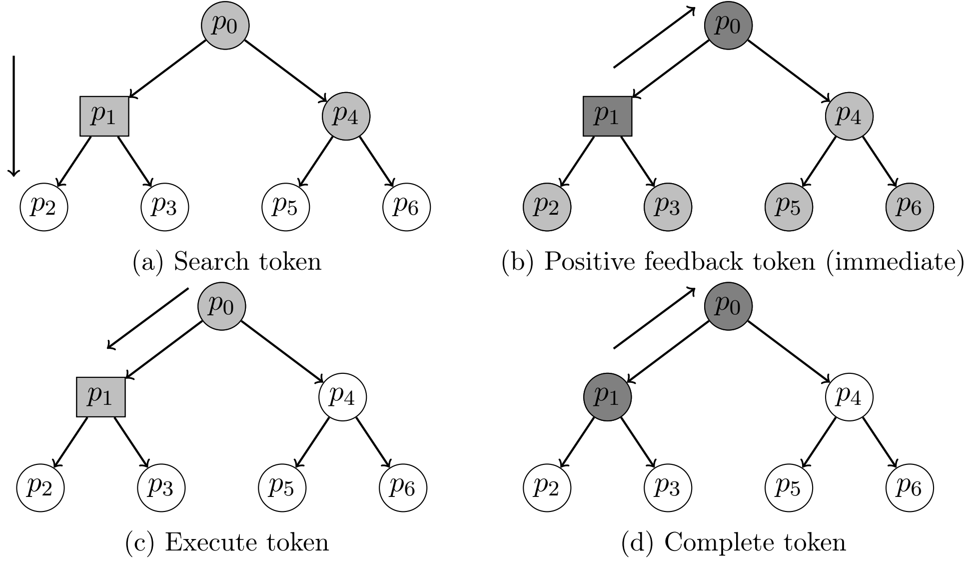
gc_1 : $\mathsf{token}_{2\text{-}a} \longrightarrow \ell := ch.\mathsf{id};\ \mathsf{x} := \neg\mathsf{x};$
gc_2 : $\mathsf{token}_{2\text{-}b} \longrightarrow \ell := -1;\ \mathsf{x} := \neg\mathsf{x};$
gc_3 : $\mathsf{token}_4 \wedge active \longrightarrow$ ***critSection()***; $\ell := -1;\ \mathsf{x} := \neg\mathsf{x};$ % Privilege
gc_4 : $\mathsf{token}_4 \wedge \neg active \longrightarrow \ell := -1;\ \mathsf{x} := \neg\mathsf{x};$

For Inner Processes

gc_5 : $\mathsf{token}_1 \wedge \neg active \longrightarrow \ell := -1;\ \mathsf{up} := \top;\ \mathsf{x} := \neg\mathsf{x};$
gc_6 : $\mathsf{token}_1 \wedge active \longrightarrow \ell := \mathsf{id};\ \mathsf{up} := \bot;\ \mathsf{x} := \neg\mathsf{x};$
gc_7 : $\mathsf{token}_{2\text{-}a} \wedge \mathsf{parent}(p).\mathsf{x} = \mathsf{x} \longrightarrow \ell := ch.\mathsf{id};\ \mathsf{up} := \bot;$
gc_8 : $\mathsf{token}_{2\text{-}b} \wedge \mathsf{parent}(p).\mathsf{x} = \mathsf{x} \longrightarrow \mathsf{up} := \bot;$
gc_9 : $\mathsf{token}_3 \wedge \mathsf{parent}(p).\ell = \mathsf{id} \wedge \ell = \mathsf{id} \longrightarrow$ ***critSection()***; $\mathsf{up} := \bot;\ \mathsf{x} := \neg\mathsf{x};$ % Privilege
gc_{10} : $\mathsf{token}_3 \wedge \mathsf{parent}(p).\ell = \mathsf{id} \wedge \exists\, ch \in \mathsf{Ch}(p) \bullet \ell = ch.\mathsf{id} \longrightarrow \mathsf{up} := \top;\ \mathsf{x} := \neg\mathsf{x};$
gc_{11} : $\mathsf{token}_3 \wedge (\mathsf{parent}(p).\ell \neq \mathsf{id} \vee \forall\, q \in \mathsf{Ch}(p) \cup \{p\} \bullet \ell \neq q.\mathsf{id}) \longrightarrow \ell := -1;\ \mathsf{up} := \bot;\ \mathsf{x} := \neg\mathsf{x};$
gc_{12} : $\mathsf{token}_4 \wedge \mathsf{parent}(p).\mathsf{x} = \mathsf{x} \longrightarrow \mathsf{up} := \bot;$

For Leaves

gc_{13} : $\mathsf{token}_1 \wedge \neg active \longrightarrow \ell := -1;\ \mathsf{x} := \neg\mathsf{x};$
gc_{14} : $\mathsf{token}_1 \wedge active \longrightarrow \ell := \mathsf{id};\ \mathsf{x} := \neg\mathsf{x};$
gc_{15} : $\mathsf{token}_3 \wedge \mathsf{parent}(p).\ell = \mathsf{id} \wedge \ell = \mathsf{id} \longrightarrow$ ***critSection()***; $\mathsf{x} := \neg\mathsf{x};$ % Privilege
gc_{16} : $\mathsf{token}_3 \wedge (\mathsf{parent}(p).\ell \neq \mathsf{id} \vee \ell \neq \mathsf{id}) \longrightarrow \mathsf{x} := \neg\mathsf{x};$



(a) Search token (b) Positive feedback token (immediate)

(c) Execute token (d) Complete token

Figure 5.2: Scenario of Algorithm 5.5. A square denotes an active process.

2. Case (2) represented by gc_{10}: if $\mathsf{parent}(p)$ points to p and p points to one of its children q, this implies that the selected process exists in the subtree rooted by q, but not q. p passes token_3, while keeping $p.\ell = q.\mathsf{id}$. Note that the children of p other than q behave as in Case 1 in the following steps because p is pointing to q.

3. Case (3) represented by $\mathsf{gc}_9, \mathsf{gc}_{15}$: if $\mathsf{parent}(p)$ points to p and p points to itself, then p is privileged to execute *critSection*(). p executes *critSection*() and switches into $p^{\perp}_{\perp}$ sending token_4 to its parent. Note that p is the unique privileged process, because by definition of trees, there exists at most one process that points to itself and is linked to the root by one path, where each parent points to its child in this path.

$\mathsf{token}_4 \uparrow$: Each process that receives token_4, forwards it to its ancestors (gc_{12}). The root finally receives token_4 which involves all its children, after the selected process executes *critSection*(). If the root is active, it is privileged, it executes *critSection*(), and it sends token_1 starting a new cycle (gc_3). Otherwise, the root simply starts a new cycle (gc_4).

Figure 5.2 illustrates an example of the two PIF cycles, where p_1 is active.

5.5 Educated Unique Process Selection

This section is structured as follows: Section 5.5.1 presents the algorithm for educated selection based on local state evaluation. Section 5.5.2 extends the algorithm to achieve selection based on evaluating global configurations.

5.5.1 Educated Selection Based on Local States

The algorithm for educated unique process selection, based on evaluating local states, is presented in Algorithm 5.6. The variables owned by each process are as follows:

- The variables up and x are chosen from Algorithm 5.4, together with the assertion $\mathsf{up} = \top$ for the root, and $\mathsf{up} = \bot$ for each leaf.
- The variable ℓ is used similar to Algorithm 5.5, however, with a difference that the value -1 does not appear in the domain of ℓ. Recall that the value -1 is used to mark that the corresponding process is not pointing to any process. The case is not needed here, because within a feedback, always one process is selected.
- The local criterion, upon which a process is selected, is abstracted by a variable $\mathsf{m} \in \mathbb{R}$. A process is selected only if the value of m of this process is the maximum among all other processes in the tree. The variable m is assumed to be updated locally and independent of the algorithm.

Besides the variables, a function *choose* : $2^{\mathcal{P}} \longrightarrow \{0, \ldots, \mathsf{n}{-}1\}$ is defined as follows: given a set of processes $\mathcal{P}'$, the function returns the id of a process that has the maximum value of m among all processes in $\mathcal{P}'$.

The stable behavior of Algorithm 5.6 is an infinite repetition of two PIF cycles, where in each cycle, the root propagates a token to all processes, and receives a feedback. Informally:

- In the first PIF cycle, the root propagates a search token that reaches all processes. Each process updates the value of m. Next, a feedback token is sent starting from the leaves and ending at the root. When a process receives a feedback token, it points to the process with the largest value of m among itself and its children, and updates the value of its m accordingly.
- The second PIF cycle is similar to the second cycle of Algorithm 5.5, except that token_3 reaches the leaves and token_4 is sent back from the leaves.

The two PIF cycles are explained in detail as follows.

First PIF Cycle

$\mathsf{token}_1 \downarrow$: the root propagates token_1. When a process p receives token_1, p updates m to a value that is independent of the algorithm, and p forwards the token to its children (gc_4), until token_1 reaches the leaves. Each leaf l_i updates $l_i.\mathsf{m}$, and switches into $l{\ddagger}i$ (gc_{10}) to send token_2 to its parent. Now each process in the tree has updated its value of m.

$\mathsf{token}_2 \uparrow$: when a process p receives token_2, p points to a process q, where $q \in \{p\} \cup \mathsf{Ch}(p)$ and q has the maximum value of m among p and its children (gc_5). Then, p copies $q.\mathsf{m}$, and switches the value of $p.\mathsf{up}$ (gc_5). Note that q might be p itself. With this action, each process p eventually points to the process with the original maximum value of m in the maximal subtree rooted by p, after copying the maximum m. Eventually, token_2 reaches the root. Next, the root starts the second PIF cycle (gc_1), after selecting a process, similarly, using the variable ℓ.

Second PIF Cycle

$\mathsf{token}_3 \downarrow$: the root sends token_3. If a process p receives token_3, one of three possible cases may exist:

Algorithm 5.6 Self-stabilizing PIF for educated unique process selection

// This represents the sub-algorithm for process p

Variables: $\mathsf{x} \in \mathbb{B}$, $\mathsf{up} \in \mathbb{B}$, $\ell \in \{0, \dots, \mathsf{n}-1\}$, $\mathsf{m} \in \mathbb{R}$

Assertions: $\mathsf{root.up} = \top \wedge \forall q \in \mathsf{Leaves} \bullet q.\mathsf{up} = \bot$

Tokens

$\mathsf{token}_1 : \mathsf{parent}(p).\mathsf{x} \neq \mathsf{x} \wedge \neg\mathsf{x}$ % Search Token
$\mathsf{token}_2 : \mathsf{up} \wedge \mathsf{x} \wedge \forall ch \in \mathsf{Ch}(p) \bullet ch.\mathsf{x} = \mathsf{x} \wedge \neg ch.\mathsf{up}$ % Feedback Token
$\mathsf{token}_3 : \mathsf{parent}(p).\mathsf{x} \neq \mathsf{x} \wedge \mathsf{x}$ % Execute Token
$\mathsf{token}_4 : \mathsf{up} \wedge \neg x \wedge \forall ch \in \mathsf{Ch}(p) \bullet ch.\mathsf{x} = \mathsf{x} \wedge \neg ch.\mathsf{up}$ % Complete Token

Functions

$update_{\mathsf{m}}()$:= $\{v \mid v \in \mathbb{R}\}$
$choose(\mathcal{P}' \subseteq \mathcal{P})$:= $\{i \mid p_i \in \mathcal{P}' \wedge \forall q \in \mathcal{P}' \bullet p_i.\mathsf{m} = \max(q.\mathsf{m})\}$
$critSection()$: Access Critical Section

Guarded Commands (gc_i : *guard* $\longrightarrow$ *action*)

For Root

$\mathsf{gc}_1 :$ $\mathsf{token}_2 \longrightarrow \ell := choose(\{p\} \cup \mathsf{Ch}(p));\ \mathsf{m} := p_\ell.\mathsf{m};\ \mathsf{x} = \neg\mathsf{x};$
$\mathsf{gc}_2 :$ $\mathsf{token}_4 \wedge \ell = \mathsf{id} \longrightarrow \boldsymbol{critSection()};\ \mathsf{m} := update_{\mathsf{m}}();\ \mathsf{x} = \neg\mathsf{x};$ % Privilege
$\mathsf{gc}_3 :$ $\mathsf{token}_4 \wedge \ell \neq \mathsf{id} \longrightarrow \mathsf{m} := update_{\mathsf{m}}(); \mathsf{x} = \neg\mathsf{x};$

For Inner Processes

$\mathsf{gc}_4 :$ $\mathsf{token}_1 \longrightarrow \mathsf{m} := update_{\mathsf{m}}();\ \mathsf{up} := \top;\ \mathsf{x} := \neg\mathsf{x};$
$\mathsf{gc}_5 :$ $\mathsf{token}_2 \wedge \neg\mathsf{token}_3 \longrightarrow \ell := choose(\{p\} \cup \mathsf{Ch}(p));\ \mathsf{m} = p_\ell.\mathsf{m};\ \mathsf{up} := \bot;$
$\mathsf{gc}_6 :$ $\mathsf{token}_3 \wedge \mathsf{parent}(p).\ell = \mathsf{id} \wedge \ell = \mathsf{id} \longrightarrow \boldsymbol{critSection()};\ \mathsf{up} := \top;\ \mathsf{x} := \neg\mathsf{x};$ % Privilege
$\mathsf{gc}_7 :$ $\mathsf{token}_3 \wedge \mathsf{parent}(p).\ell = \mathsf{id} \wedge \exists q \in \mathsf{Ch}(p) \bullet p.\ell = q.\mathsf{id} \longrightarrow \mathsf{up} := \top;\ \mathsf{x} := \neg\mathsf{x};$
$\mathsf{gc}_8 :$ $\mathsf{token}_3 \wedge (\mathsf{parent}(p).\ell \neq \mathsf{id} \vee \forall q \in \mathsf{Ch}(p) \bullet p.\ell \neq q.\mathsf{id}) \longrightarrow \ell := \mathsf{id};\ \mathsf{up} := \top;\ \mathsf{x} := \neg\mathsf{x};$
$\mathsf{gc}_9 :$ $\mathsf{token}_4 \wedge \neg\mathsf{token}_1 \longrightarrow \mathsf{up} := \bot;$

For Leaves

$\mathsf{gc}_{10} :$ $\mathsf{token}_1 \longrightarrow \mathsf{m} := update_{\mathsf{m}}();\ \ell := \mathsf{id};\ \mathsf{x} := \neg\mathsf{x};$
$\mathsf{gc}_{11} :$ $\mathsf{token}_3 \wedge \mathsf{parent}(p).\ell = \mathsf{id} \longrightarrow \boldsymbol{critSection()};\ \mathsf{x} := \neg\mathsf{x};$ % Privilege
$\mathsf{gc}_{12} :$ $\mathsf{token}_3 \wedge \mathsf{parent}(p).\ell \neq \mathsf{id} \longrightarrow \mathsf{x} := \neg\mathsf{x};$

Algorithm 5.7 Extension of Algorithm 5.6

Additional Variables

$snapShot = [k_0, \dots, k_{\mathsf{n}-1}]$, where $k_i \in \mathbb{R}$ for $0 \leq i \leq \mathsf{n}-1$

Extended Functions

$update_{\mathsf{m}}()([k_0, \dots, k_{\mathsf{n}-1}]) = \{v \in \mathbb{R} \mid v \text{ is dependent on } [k_0, \dots, k_{\mathsf{n}-1}]\}$

Extended Guarded Commands $(2, 3, 4, 6, 9, 10, 11)$

$\mathsf{gc}_{2'} : \dots \longrightarrow \boldsymbol{critSection()};\ snapShot.k_{\mathsf{id}} = k;\ \mathsf{m} := update_{\mathsf{m}}(snapShot);\ \mathsf{x} = \neg\mathsf{x};$
$\mathsf{gc}_{3'} : \dots \longrightarrow snapShot = p_\ell.snapShot;\ \mathsf{m} := update_{\mathsf{m}}(snapShot);\ \mathsf{x} := \neg\mathsf{x};$
$\mathsf{gc}_{4'} : \dots \longrightarrow snapShot = \mathsf{parent}(p).snapShot;\ \mathsf{m} := update_{\mathsf{m}}(snapShot);\ \mathsf{up} := \top; \mathsf{x} := \neg\mathsf{x};$
$\mathsf{gc}_{6'} : \dots \longrightarrow \boldsymbol{critSection()};\ snapShot.k_{\mathsf{id}} := k;\ \mathsf{up} := \bot;\ \mathsf{x} := \neg\mathsf{x};$
$\mathsf{gc}_{9'} : \dots \longrightarrow snapShot := p_\ell.snapShot;\ \mathsf{up} := \bot;$
$\mathsf{gc}_{10'} : \dots \longrightarrow snapShot = \mathsf{parent}(p).snapShot;\ \mathsf{m} := update_{\mathsf{m}}(snapShot);\ \ell := \mathsf{id};\ \mathsf{x} := \neg\mathsf{x};$
$\mathsf{gc}_{11'} : \dots \longrightarrow \boldsymbol{critSection()};\ snapShot.k_{\mathsf{id}} = k;\ \mathsf{m} := update_{\mathsf{m}}(snapShot);\ \mathsf{x} := \neg\mathsf{x};$

- Case (1) represented by $\mathsf{gc}_8, \mathsf{gc}_{12}$: if $\mathsf{parent}(p)$ is not pointing to p, or p is neither pointing to itself nor to one of its children (if p is not a leaf), then there is no selected process in the maximal subtree rooted by p. p sets ℓ to $p.\mathsf{id}$, to prohibit any child from executing *critSection*() after forwarding token_3. Note that the children of p (if they exist) behave similarly in the next step. The leaves do not need to change the value of ℓ, since they have no children.

- Case (2) represented by gc_7: if $\mathsf{parent}(p)$ points to p and p points to one of its children q, this implies that the selected process exists in the maximal subtree rooted by q. p passes token_3, while keeping $p.\ell = q.\mathsf{id}$. Note that the children of p other than q react as in Case (1).

- Case (3) represented by $\mathsf{gc}_6, \mathsf{gc}_{11}$: if $\mathsf{parent}(p)$ points to p and p points to itself, then p has a privilege. p executes *critSection*() and forwards the token. Note that p has a unique privilege, because by definition of tree, there is at most one process that points to itself and is linked to the root by one path, where each parent points to its child.

$\mathsf{token}_4 \uparrow$: Next, token_4 is forwarded to the root (gc_9). The root receives token_4 which involves all its children, after any selected non-root process executes *critSection*(). If the selected process is the root, then gc_2 is enabled, the root executes *critSection*(), and propagates token_1 to its children starting a new PIF cycle. Otherwise (gc_3), the root simply starts a new PIF cycle.

Figure 5.3 illustrates an example, where each process is labelled by its value of m. p_3 has the highest value of m, which equals 20.

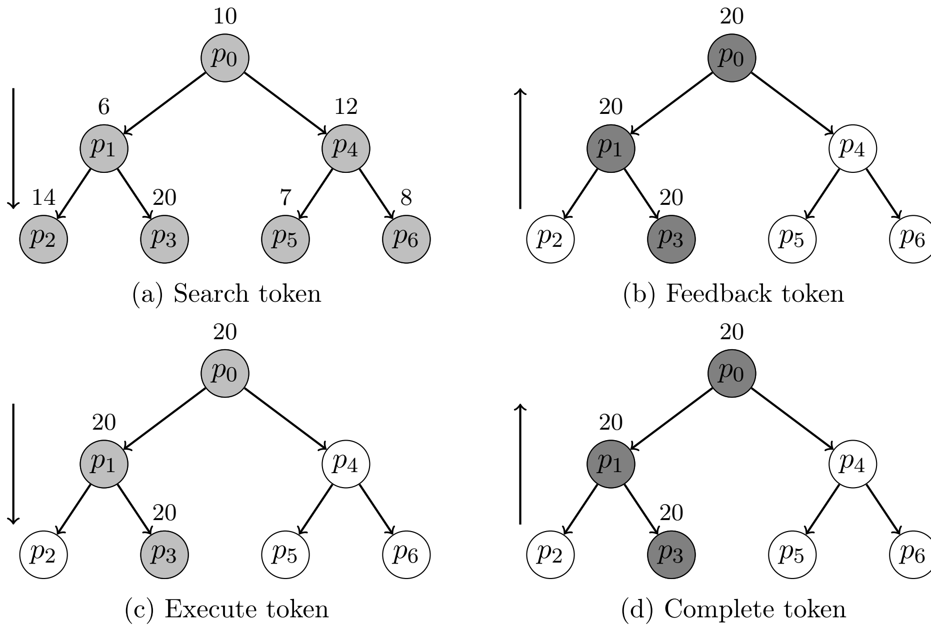


Figure 5.3: Scenario of Algorithm 5.6. p_3 has the highest value of m.

5.5.2 Educated Selection Based on Global Configurations

Algorithm 5.6 is extended to satisfy educated selection based on the topology configuration; i.e. a process is selected based on a global view to the states of all processes. For simplicity, only the extension of Algorithm 5.6 is shown. It is presented in Algorithm 5.7.

In Algorithm 5.6, whenever m is updated by the function $update_{\mathsf{m}}()$, the function returns values that are independent of the algorithm, and are based on local criterion belonging to the owning process. In the extended version, the value of m is updated according to the global configuration of the topology. This raises the need that each process needs to know global information.

The global information is abstracted by the vector *snapShot*, which is owned by each process, and is defined as follows:

$$snapShot = [k_0, \ldots, k_{\mathsf{n}-1}],$$

where $k_i \in \mathbb{R}$, for $0 \leq i \leq \mathsf{n}-1$, is the relevant evaluation of the local state of p_i. Each process p updates $p.\mathsf{m}$ according to the value of $p.snapShot$. Algorithm 5.6 is extended by changing the commands $\mathsf{gc}_{2\text{-}4}, \mathsf{gc}_6, \mathsf{gc}_{9\text{-}11}$ (cf. Algorithm 5.7). With this extension, the stable behavior of the algorithm is as follows: in the first PIF cycle, when a process receives token_1, it copies the parent's snapshot, and updates its value of m according to the snapshot ($\mathsf{gc}_{4'}, \mathsf{gc}_{10'}$). With this action, a copy of the snapshot reaches each process. Next, the remainder of the first PIF cycle continues normally. In the second PIF cycle, the root sends token_3 that reaches the selected process p. After p runs $critSection()$, it modifies the snapshot based on its current value of k ($\mathsf{gc}_{6'}, \mathsf{gc}_{11'}$). Next, the parent of p copies the new snapshot, and forwards it to the root ($\mathsf{gc}_{9'}$). The root handles the snapshot similar to the other processes ($\mathsf{gc}_{2'}, \mathsf{gc}_{3'}$).

Note that the above behavior represents a stable behavior; the snapshot sent by the root matches the values of k of all processes. If the snapshot contains an incorrect value of some k, the snapshot is called *inconsistent*. Intuitively, inconsistent snapshots are required to be corrected. In the following, a correction method for snapshot inconsistency is sketched.

To correct snapshot inconsistency, the notion of *highlighting* a snapshot is used. A snapshot *snap* of a process p is said to be *highlighted* iff *snap* contains at least one *null* value of some k; i.e. iff

$$p.snapShot = [k_0, \ldots, null, \ldots, k_{\mathsf{n}-1}].$$

Additionally, *snap* is called *empty* if it contains only *null* values; i.e. iff

$$p.snapShot = [null, \ldots, null].$$

The snapshot inconsistency is corrected in the first PIF cycle as follows:

1. When the root propagates token_1 while having an inconsistent *snapShot*, then there exists a process p_j such that $p_j.k$ is not equal to $snapShot.k_j$. Eventually, p_j receives token_1.

2. When p_j copies its parent's snapshot, p_j checks if there is snapshot inconsistency, or if $\mathsf{parent}(p).snapShot$ is empty. If so, p_j sets its snapshot as empty; i.e.

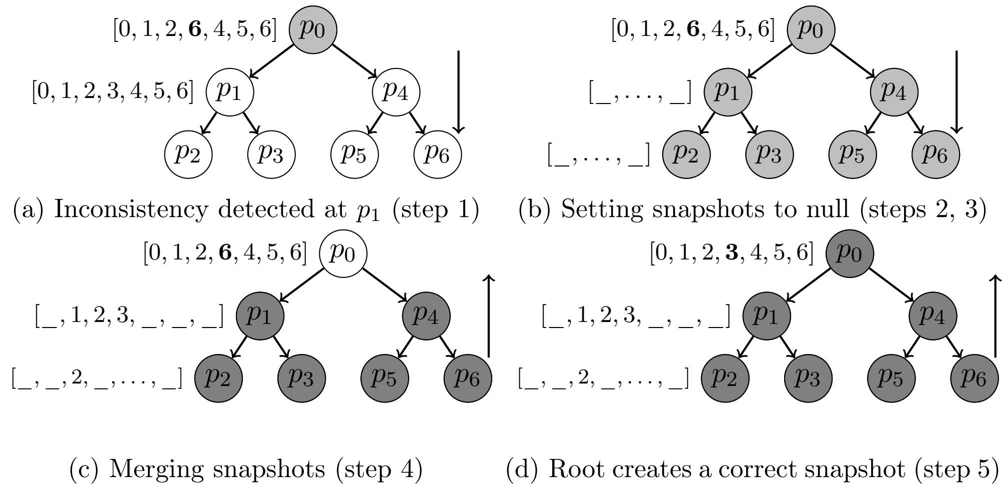
$$p_j.snapShot = [null, \ldots, null].$$

3. All processes in the maximal subtree rooted by p_j set their snapshots to empty, analogous to step 2, since token_1 reaches every process.

4. Now starting from the leaves, for each process p that receives token_2, if p recognizes a highlighted snapshot in one of its children or itself, then p creates a new snapshot by merging the snapshots of its children, and adding its value of k. With this action, the snapshot of p contains correct values of all processes in the maximal subtree rooted by p, and *null* values for the processes that are not in the subtree.

5. Once the root receives a feedback token (token_2), if the root recognizes snapshot inconsistency or a highlighted snapshot in one of its children, the root sets its snapshot empty, merges it with the highlighted snapshots of its children, and then, the root adds the missing variables from any non-highlighted snapshot. Now the root has a correct snapshot, that is propagated in the next PIF cycle.

Figure 5.4 illustrates an example. In this example, for simplicity, a correct snapshot records the id i of each process as a value of k_i. In Figure 5.4a, p_1 detects an inconsistency in the snapshot sent by the root. In Figure 5.4b, p_1, p_2, and p_3 set their snapshots to empty ("_" denotes *null*.) Next in Figure 5.4c, p_1, p_2, and p_3 add their own values and merge the values of their children. Finally, in Figure 5.4d, the root creates a correct snapshot.



(a) Inconsistency detected at p_1 (step 1) (b) Setting snapshots to null (steps 2, 3)

(c) Merging snapshots (step 4) (d) Root creates a correct snapshot (step 5)

Figure 5.4: Snapshot correction. The symbol "_" denotes *null*.

5.6 Correctness and Time Complexity

This section presents the correctness and time complexity proofs for Algorithm 5.5 and Algorithm 5.6. The critical part of the correctness is the issue of granting a privilege,

which is formalized in Definition 4.5 of Section 4.4 using the condition *priv*. The condition *priv* over a configuration is defined as: there is a process that is privileged. In this section, concerning Algorithm 5.5, a process is privileged iff one of the commands $\mathsf{gc}_4, \mathsf{gc}_9$, and gc_{15} is enabled. Concerning Algorithm 5.6, a process is privileged iff one of the commands $\mathsf{gc}_2, \mathsf{gc}_6$, and gc_{11} is enabled.

To show the correctness of Algorithm 5.5 and Algorithm 5.6, the following environment is set: The corresponding algorithm is executed over a topology $\mathcal{T} = (\mathcal{P}, \mathcal{E})$ using the shared memory model. The scheduler is synchronous for Algorithm 5.5, and asynchronous for Algorithm 5.6, where the asynchronous one is more general. The time complexity and recurrence are shown for the synchronous scheduler.

The properties to be shown are as follows:

- Both algorithms are self-stabilizing wrt. *UPS* in $\mathsf{depth}(\mathcal{T})$ steps.
- Let $\Delta = 1/(4 \cdot \mathsf{depth}(\mathcal{T}))$. Algorithm 5.5 is self-stabilizing wrt. $priv_\Delta$ in $7 \cdot \mathsf{depth}(\mathcal{T})$ steps.
- Concerning Algorithm 5.6, in $3 \cdot \mathsf{depth}(\mathcal{T})$ steps
  - educated unique process selection holds.
  - within any two subsequent PIF cycles, exactly one process is granted a privilege.

In the following, some terms are fixed, and some basic lemmata for both algorithms are presented. First, the notions of *top* and *bottom tokens* are defined.

Definition 5.3 (Top, Bottom Token)**.** A process p in a topology is said to have a **top token** in a configuration γ iff $p \neq \mathsf{root} \wedge p.\mathsf{x} \neq \mathsf{parent}(p).\mathsf{x}$. Process p is said to have a **bottom token** in γ iff p has an enabled command and $p.\mathsf{x} = \mathsf{parent}(p).\mathsf{x}$. ♢

Corollary 5.1. Each process cannot have both, top and bottom tokens in the same configuration.

Proof. By definition of the commands, a process p has a top token iff $\mathsf{parent}(p).\mathsf{x} \neq p.\mathsf{x}$, and p may have a bottom token only if $\mathsf{parent}(p).\mathsf{x} = p.\mathsf{x}$, which implies that p cannot have both tokens in the same configuration. □

In a legitimate configuration, it is supposed to be that the tokens and the values of ℓ should result from the actions of the commands. However, in an illegitimate configuration, there might be arbitrary tokens and values. The following three lemmata concern these token issues. The lemmata apply under the asynchronous scheduler, which implies that they also apply under the synchronous scheduler. The time complexity measures regard the synchronous scheduler.

Lemma 5.1. Let $p_1, \ldots, p_s$ be a path in the topology. Let $b \in \mathbb{B}$, and let γ_0 be a configuration in which $p_{i_b}^{?}$ holds for $1 \leq i \leq s$. For each execution $\gamma_0, \gamma_1, \ldots,$ each of $p_2, \ldots, p_s$ has no top token until p_1 changes its x value through: a bottom token if $p_1 = \mathsf{root}$, or a top token.

PROOF. A process p may have a top token only, if $p.\mathsf{x} \neq \mathsf{parent}(p).\mathsf{x}$. In γ_0, for $2 \leq j \leq \mathsf{n}$, $p_j.\mathsf{x} = \mathsf{parent}(p_j).\mathsf{x} = b$, and therefore each p_j does not have a top token in γ_0. p_j may have a top token in any following configuration only if $\mathsf{parent}(p_j)$ changes the value of x. Bottom tokens may exist. However, execution steps following bottom tokens do not change the value of x for all processes except for the root. By tree definition, the root cannot be any of processes p_j. The only process that may change the value of x is p_1: (i) if p_1 is the root, then it switches into ${p_1}^?_{\neg b}$ through a bottom token, (ii) otherwise, p_1 may switch into ${p_1}^?_{\neg b}$ by only a top token. □

LEMMA 5.2. Given a path $p_0, \ldots, p_s$ in a topology, and a configuration γ_0 in which ${p_0}^?_b$ for $b \in \mathbb{B}$. For each execution $\gamma_0, \gamma_1, \ldots,$ there exists finally a configuration γ_u, where $0 \leq u \leq s-1$ for the synchronous scheduler, such that: if $\forall\, i \in [0, u] \bullet \gamma_i \models {p_0}^?_b$, then $\forall\, j \in [0, u] \bullet \gamma_u \models {p_j}^?_b$; i.e. if ${p_0}^?_b$ holds, then all other processes p_j switch into ${p_j}^?_b$ in at most u steps.

PROOF. If for all $0 \leq j \leq s$, ${p_j}^?_b$ holds in γ_0, then the claim holds with $u = 0$. Otherwise, consider the process p_e with the least depth among $p_1, \ldots, p_s$ and in state ${p_e}^?_{\neg b}$. By Lemma 5.1, p_{e-1} has no bottom token, and p_e has a top token. In the step (γ_0, γ_1), p_e switches into ${p_e}^?_b$. Now in γ_1, p_e does not have a top token. Note that x is equal to b in all the processes $p_0, \ldots, p_e$. Inductively, the same procedure applies to the processes $p_{e+1}, p_{e+2}, \ldots$ in the next steps, respectively, unless p_0 switches its value of x. Since the number of processes is s, after at most $u = s$ synchronous steps, each process has $\mathsf{x} = b$, given that ${p_0}^?_b$ holds. □

LEMMA 5.3. Let d be the depth of a topology, and let γ_0 be a configuration in which $\mathsf{root}^\top_b$ for $b \in \mathbb{B}$. For each execution $\gamma_0, \gamma_1, \ldots,$ there exists finally a configuration γ_u, such that $\gamma_u \models \mathsf{root}^\top_{\neg b}$, and $1 \leq u \leq 2d$ for the synchronous scheduler.

PROOF. By Lemma 5.2, after at most d synchronous steps at a configuration γ_s, for each process p, $p^?_b$ holds, and in particular, $p^\perp_b$ for the leaves, if $\mathsf{root}^\top_b$ holds in each $\gamma_0, \ldots \gamma_s$. In γ_s, there is no top token for any process in the tree because x is equal to b for all processes. Moreover, by Lemma 5.1, steps following bottom tokens do not enable top tokens unless the root changes its configuration. Let p be a process with the largest depth such that $p^\top_b$ holds in γ_s. Since for all leaves $\mathsf{up} = \perp$, the depth of p is at most $d-1$. In addition, every child ch of p is in a configuration $ch^\perp_b$. In the next step, by definition of the commands – $\mathsf{gc}_{1-4}, \mathsf{gc}_{7-8}, \mathsf{gc}_{12}$ for Algorithm 5.5 and $\mathsf{gc}_{1-3}, \mathsf{gc}_5, \mathsf{gc}_9$ for Algorithm 5.6 – at least one of them is enabled, and each switches $p.\mathsf{up}$ into $\perp$. Analogously, in γ_{s+1}, each process p_r at the same depth of p is in a state ${p_r}^\perp_b$. Note that in γ_{s+1}, there are no top or bottom tokens for all processes in any subtree rooted by any process at depth $d-1$. Inductively, the processes in $d-2$ and lower depths perform the same action (if required) in the next steps, respectively. After at most $d-1$ steps of γ_s, a configuration γ_{u-1} is reached where one of the commands gc_{1-4} for Algorithm 5.5 and gc_{1-3} for Algorithm 5.6 is enabled. In (γ_{u-1}, γ_u), the root switches into $\mathsf{root}^\top_{\neg b}$. By summing up, the overall number of steps u may reach up to: $d + (d-1) + 1 = 2d$. □

5.6.1 Correctness of Algorithm 5.5

This part concerns the correctness of Algorithm 5.5 under the synchronous scheduler. In the following parts, a legitimate configuration is defined. Next, the convergence wrt. a legitimate configuration is shown. After that, the closure of legitimate configurations and recurrence properties are considered. Finally, self-stabilization wrt. unique process selection is shown.

Legitimate Configuration

This part specifies what a legitimate configuration is. Intuitively, a legitimate configuration is defined based on the desired behavior, which in turn satisfies the desired properties, basically unique process selection. Note that the definition and proofs are built upon rigorous theory, due to the algorithms' nature of having multiple properties over configurations and executions.

Roughly, a legitimate configuration should guarantee the following properties – formalized in Definition 5.7:

1. If a process p sends a search token ($p^{\top}_{\top}$), $p.\ell$ = -1 holds, and if p is a non-root process, then p is not active.
2. If a process p sends a feedback token ($p^{\perp}_{\top}$):
   - either there exists a path leading to an active process, where each process in this path points to the next one. Such a path is called an *active path*, denoted by *aPath*.
   - Or $p.\ell$ = -1, and there exists no active process in the subtree rooted by p.
3. If the root sends a search token ($\mathsf{root}^{\top}_{\top}$):
   - $\mathsf{root}.\ell$ = -1.
   - No process is granted a privilege
4. If the root sends an execute token ($\mathsf{root}^{\top}_{\perp}$):
   - Either there exists exactly one path, which is called execution path (*ePath*), that links the root to an active process, where each parent points to its child by the variable ℓ, or
   - there exists no active process in the tree, and no process other than the root sends an execute token or is privileged.

Since it is assumed that a process becomes active independent of the algorithm, it is needed to distinguish between being active while receiving a search token, and switching to active after receiving a search token. Hence, the term *last active* is defined.

Definition 5.4. Given an execution $\gamma_0, \gamma_1, \ldots$ over a topology, a process p is said to be **last active** in a configuration γ_i, denoted by $p.l\text{-}active_{\gamma_i}$, iff p is *active* in a configuration γ_j where $j \leq i$, and γ_j is the last configuration in which p received token_1. ♢

Definition 5.5. Given a configuration γ, an **active path** at γ, denoted by $aPath_\gamma$, is a path $p_r, \ldots, p_s$ for $r \leq s$ such that γ is defined as follows:

$$\forall\, r \leq j \leq s \bullet p_j{}^{\perp}_{\top} \;\wedge\; p_s.l\text{-}active \;\wedge\; p_s.\ell = p_s.\mathsf{id} \wedge$$
$$\forall\, r \leq i < s \bullet p_i.\ell = p_{i+1}.\mathsf{id} \;\wedge\; \neg p_i.l\text{-}active \qquad \diamondsuit$$

Definition 5.6. Given a configuration γ, an **execution path** at γ, denoted by $ePath_\gamma$, is a path $p_0, \ldots, p_s$ such that $p_0 = \mathsf{root}$, and γ is defined as follows: for $0 \leq i < s$:

$$p_i.\ell = p_{i+1}.\mathsf{id} \wedge p_s.\ell = p_s.\mathsf{id} \wedge p_s{}^{\perp}_{?} \wedge (p_i{}^{\top}_{?} \longrightarrow p_i{}^{\top}_{\perp}) \wedge$$
$$(p_{i+1}{}^{\top}_{\perp} \longrightarrow p_i{}^{\top}_{\perp}) \wedge (p_{i+1}{}^{?}_{\perp} \longrightarrow p_i{}^{?}_{\perp}) \wedge (p_i{}^{\perp}_{?} \longrightarrow p_{i+1}{}^{\perp}_{?}) \wedge$$
$$(p_i{}^{\perp}_{\top} \longrightarrow \neg p_i.l\text{-}active) \wedge (p_s{}^{\perp}_{\top} \longrightarrow p_s.l\text{-}active) \wedge$$
$$\forall\, p \notin \{p_0, \ldots, p_s\} \bullet (p^{\top}_{?} \longrightarrow p^{\top}_{\top}\text{-}1) \wedge (p^{?}_{\top} \wedge \mathsf{parent}(p)^{?}_{\perp} \longrightarrow \mathsf{parent}(p).\ell \neq p.\mathsf{id}) \qquad \diamondsuit$$

Definition 5.7. A **legitimate configuration**, abbreviated by *legConfig*, for Algorithm 5.5 is a configuration γ of a topology $\mathcal{T} = (\mathcal{P}, \mathcal{E})$ that satisfies:

$$\text{1- } \forall\, p \in \mathcal{P} \bullet (p^{\top}_{\top} \longrightarrow p^{\top}_{\top}\text{-}1 \wedge (p \neq \mathsf{root} \longrightarrow \neg p.l\text{-}active)) \tag{5.1}$$

$$\text{2- } \forall\, p_s \in \mathcal{P} \bullet p_s{}^{\perp}_{\top} \longrightarrow$$
$$\exists\, r \geq s \bullet p_s, \ldots, p_r : aPath_\gamma \;\vee \tag{5.2}$$
$$\forall\, path : p_s, \ldots, p_z \bullet \forall\, e \in [s, z] \bullet \neg p_e.l\text{-}active \wedge p_e{}^{\perp}_{\top}\text{-}1 \tag{5.3}$$

$$\text{3- } \mathsf{root}^{\top}_{\top} \longrightarrow$$
$$\forall\, p \in \mathcal{P} \bullet p^{\top}_{?} \longrightarrow p^{\top}_{\top}\text{-}1 \;\wedge \tag{5.4}$$
$$\forall\, p \neq \mathsf{root} \bullet (p^{?}_{\top} \wedge \mathsf{parent}(p).\ell = p.\mathsf{id}) \longrightarrow \mathsf{parent}(p)^{?}_{\top} \tag{5.5}$$

$$\text{4- } \mathsf{root}^{\top}_{\perp} \longrightarrow$$
$$\exists\, p \in \mathcal{P} \bullet p.l\text{-}active \longrightarrow \exists !\, ePath_\gamma \wedge {}^{1} \tag{5.6}$$
$$\neg \exists\, p \in \mathcal{P} \bullet p.l\text{-}active \longrightarrow \forall\, p \neq \mathsf{root} \bullet$$
$$p^{\top}_{?} \longrightarrow p^{\top}_{\top}\text{-}1 \wedge \tag{5.7}$$
$$p^{?}_{\top} \wedge \mathsf{parent}(p)^{?}_{\perp} \longrightarrow \mathsf{parent}(p).\ell \neq p.\mathsf{id} \tag{5.8}$$

The set of legConfigs for Algorithm 5.5 is denoted by Γ_{leg}. ♢

Note that legConfig is used without mentioning Algorithm 5.5, as it is clear from the context.

Convergence wrt. a LegConfig

Theorem 5.1 (Convergence). For each execution $\gamma_0, \gamma_1, \ldots$ over a topology of depth d, there exists finally a legConfig $\gamma_u \in \Gamma_{leg}$, such that $0 \leq u \leq 3d$.

Proof Sketch (the full proof is given in Appendix A.1). Lemma 5.3 states that the root changes its configuration from $\mathsf{root}^{\top}_{b}$ into $\mathsf{root}^{\top}_{\neg b}$ in $2d$ steps. Further analysis shows that a legConfig is reached after d steps from the last change of the root's configuration. This sums up to to $d + 2d = 3d$. □

[1] The symbol ∃! denotes “there exists exactly one”.

Note that an execution may require up to $3d$ steps to reach a legConfig if a leaf process was initially sending a positive feedback token ($\mathsf{token}_{2\text{-}a}$) while not being active, and while there are other processes that still did not receive the search token.

Closure

To prove the closure of the set of legConfigs, the set is divided into four categories. Next, it is shown that any execution goes only through the categories, such that there exist time bounds for which the execution stays in one category. When a process is privileged, the following step – in which the process executes *critSection*() – moves between two particular categories. Using this scheme, the recurrence properties are proven as well.

The legConfigs are categorized into the following four categories – formalized in Definition A.1:

- Category 1 contains the legConfigs in which the root is in a state $\mathsf{root}^{\top}_{\top}$.
- Category 2 (resp. Category 3) contains all legConfigs in which the root is in a state $\mathsf{root}^{\top}_{\bot}$ and there exists an active process that still has not executed *critSection*() (resp. has executed).
- Category 4 includes all legConfigs in which the root is in a state $\mathsf{root}^{\top}_{\bot}$ and there is no active process.

A system executes through legConfigs wrt. the defined categories as follows – cf. Figure 5.5:

- Starting from a legConfig in Category 1, any execution eventually reaches a legConfig in Category 2 if there exists an active non-root process, or in Category 4 if there exists no active non-root process, only through legConfigs from Category 1. No process executes *critSection*() while $\mathsf{root}^{\top}_{\top}$ holds.
- Starting from any legConfig in Category 2, any execution reaches a legConfig in Category 3 through legConfigs in Category 2. The command *critSection*() is executed by only one active process.
- From any legConfig in Category 3 (resp. Category 4), any execution reaches a legConfig in Category 1 through legConfigs in Category 3 (resp. 4), where only the root may execute *critSection*().

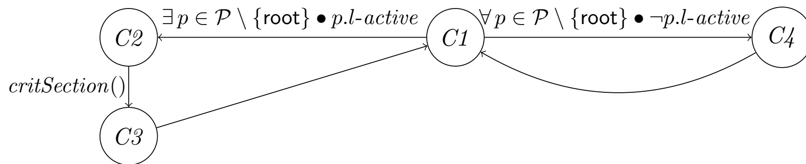


Figure 5.5: Categories of legitimate configurations for Algorithm 5.5

Theorem 5.2 (Closure)**.** For each execution $\gamma_0, \gamma_1, \ldots$ over a topology, if $\gamma_0 \in \Gamma_{leg}$, then for $i \geq 0$, $\gamma_i \in \Gamma_{leg}$.

Proof. The full proof is found in Appendix A.2. □

Unique Process Selection

This part shows that Algorithm 5.5 guarantees self-stabilization wrt. *UPS* in $\mathsf{depth}(\mathcal{T})$ steps.

Theorem 5.3. Given a configuration $\gamma_0 \in \Gamma$ of a topology of depth d, for each execution $\gamma_0, \gamma_1, \ldots,$ unique process selection is satisfied in each configuration $\gamma_d, \gamma_{d+1}, \ldots$.

Proof Sketch (the full proof is given in Appendix A.3). The liveness property of *UPS* holds following Figure 5.5. If the safety property of *UPS* is violated in some configuration, this is due to having at least one execute token, and either one execute token or the root is privileged. In any case, in $\mathsf{depth}(\mathcal{T})$ steps, there remains at most one token, which grants a privilege to one process, since execute tokens are initiated by the root and directed to one process. □

Recurrence Properties

The recurrence of *priv*, achieved by Algorithm 5.5, is $\Delta = 1/(4 \cdot \mathsf{depth}(\mathcal{T}))$, and the convergence time to achieve it ($priv_\Delta$-convergence time) is $7 \cdot \mathsf{depth}(\mathcal{T})$ steps.

Theorem 5.4. Let $\Delta = 1/(4 \cdot \mathsf{depth}(\mathcal{T}))$. Algorithm 5.5 guarantees $priv_\Delta$-convergence in $7 \cdot \mathsf{depth}(\mathcal{T})$.

Proof Sketch (the full proof is given in Appendix A.4). Let d be $\mathsf{depth}(\mathcal{T})$. Algorithm 5.5 is self-stabilizing wrt. a legConfig in $3d$ steps, and is self-stabilizing wrt. *UPS* in d steps. By thorough analysis of the closure property (cf. Figure 5.5), a process is granted a privilege in at most $4d$ steps after the algorithm stabilizes to a legConfig. This sums up to $7d$ steps. □

5.6.2 Correctness of Algorithm 5.6

This part concerns the correctness of Algorithm 5.6. The part is structured similar to Section 5.6.1: a legitimate configuration for Algorithm 5.6 is defined. Next, the convergence wrt. legConfig, closure, recurrence properties, and educated unique process selection are considered.

Legitimate Configuration and its Properties

In Algorithm 5.6, the value returned by $update_{\mathsf{m}}()$ is independent of the algorithm, and is stored in the variable m. Since m is copied from a process after receiving token_2 ($\mathsf{gc}_1, \mathsf{gc}_5$), it is needed to distinguish between the copied value, and the returned one by $update_{\mathsf{m}}()$. The notation μ_i is used to denote the value returned by the last call of $update_{\mathsf{m}_i}()$ by a process p_i.

Definition 5.8. Given a process p_i, $\boldsymbol{\mu_i}$ is the value returned by the last call of $update_{\mathsf{m}_i}()$ by p_i. ♢

Definition 5.9. A **legitimate configuration**, abbreviated by *legConfig*, for Algorithm 5.6 is a configuration γ of a topology $\mathcal{T}$ that satisfies the following:

1. Let $b \in \mathbb{B}$. For each process p:
   a) if $p_b^\top$ holds, then for each process p' in the path that is linked from the root to p, $p'^\top_b$ holds.
   b) if $p_b^\perp$ holds, then for each process p'' in the maximal subtree rooted by p, $p''^\perp_b$ holds.
2. For each process p_j in state $p_j{}^\top_\top$, $\mathsf{m}_j = \mu_j$.
3. For each process p_j in state $p_j{}^\perp_\top$, there exists a process p_s in the maximal subtree ($\mathcal{T}'$) rooted by p_j, such that $p_j, \ldots, p_s$ is a path, and for all $j \le i < s$, $p_i.\ell = i+1$, $p_s.\ell = s$, m is equal among all processes in $\mathcal{T}'$, $\mathsf{m}_s = \mu_s$, and μ_s is the maximum among all processes in $\mathcal{T}'$.
4. Let p_0 be the root. If $p_0{}^\top_\perp$, then
   a) either $p_0.\ell = 0$, and $\mathsf{m}_0 = \mu_0$ is the maximum among all processes, or
   b) there exists a path $p_0, \ldots, p_s$ such that for each process p_i, where $0 \le i < s$, $p_i.\ell = i+1$, $p_s.\ell = s$, and $\mathsf{m}_s = \mu_s$ is the maximum among all processes in $\mathcal{T}$.

   for each non-root process p_j in $\mathcal{T}$, where p_j is not in the path $p_0, \ldots, p_s$ (if exist), if $p_j{}^\top_\perp$, then $p_j.\ell = j$.
5. There is at most one process p_j, such that $\mathsf{token}_3 \wedge \mathsf{parent}(p_j).\ell = j \wedge p_j.\ell = j$ holds for p_j.

The set of legConfigs for Algorithm 5.6 is denoted by Γ_{leg}. ◇

Note that in this section, when "legConfig" is mentioned, it is naturally considered wrt. Algorithm 5.6.

Convergence wrt. a legConfig

The convergence proof holds for the asynchronous scheduler following [Dij74, Dij86, Kru79]. For the synchronous scheduler, the proof idea and time complexity is the same as in the case of Algorithm 5.5 (Theorem 5.1). Therefore, only a proof sketch is provided.

Theorem 5.5 (Convergence)**.** For each execution $\gamma_0, \gamma_1, \ldots$ over a topology, there exists finally a legConfig $\gamma_j \in \Gamma_{leg}$, such that $0 \le j \le 3 \cdot \mathsf{depth}(\mathcal{T})$.

Proof Sketch. Lemma 5.3 states that for each execution, the root changes its state from $\mathsf{root}_b^\top$ into $\mathsf{root}_{\neg b}^\top$ in $2 \cdot \mathsf{depth}(\mathcal{T})$ steps. Further analysis (similar to the proof of Theorem 5.1) show that a legConfig is reached in $\mathsf{depth}(\mathcal{T})$ steps after the root changes its state. The sum is equal to $\mathsf{depth}(\mathcal{T}) + 2 \cdot \mathsf{depth}(\mathcal{T}) = 3 \cdot \mathsf{depth}(\mathcal{T})$. □

Note that an execution may require up to $3 \cdot \mathsf{depth}(\mathcal{T})$ steps to reach a legConfig if a leaf process was initially having a fake value of m being the maximum among all other processes, while there are other processes that still did not update their values of m.

Closure

It is shown that given a topology, each execution with an initial legConfig does not reach a non-legConfig. Similar to Section 5.6.1, the legConfigs are categorized into four categories – cf. Figure 5.6:

- Category 1 – *C1* – contains all legConfigs in which the root is in a state $\mathsf{root}_{\top}^{\top}$.
- *C2* (resp. *C3*) contains all legConfigs in which the root is in a state $\mathsf{root}_{\bot}^{\top}$, and there exists a selected non-root process that still has not (resp. has already) executed *critSection*().
- *C4* includes all legConfigs in which the root is the selected process.

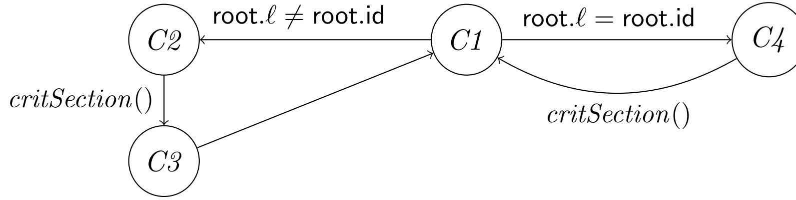


Figure 5.6: Categories of legitimate configurations for Algorithm 5.6

Theorem 5.6 (Closure)**.** For each execution $\gamma_0, \gamma_1, \ldots$ over a topology: if $\gamma_0 \in \Gamma_{leg}$, then for $i \geq 0$, $\gamma_i \in \Gamma_{leg}$.

Proof Sketch (the full proof is given in Appendix A.5). For each configuration γ_i in any of the categories *C1*, *C2*, *C3*, *C4*, for each execution step (γ_i, γ_{i+1}), γ_{i+1} is also in one of the categories, following Figure 5.6. The union of the four categories is the set of all legConfigs. This implies the theorem. □

Recurrence Properties

This part considers performance properties of Algorithm 5.6. In this case, the $priv_{\Delta}$-convergence time is not considered for the following reason: In general, Algorithm 5.6 guarantees that *priv* holds once every two subsequent PIF cycles. However, depending on the depth of the selected processes, a process at depth d may be privileged in configuration γ_i, and the next selected process at a higher depth, e.g. $d + r$, is privileged in configuration $4 \cdot \mathsf{depth}(\mathcal{T}) + r$. This indicates that $priv_{\Delta}$ may not necessarily hold. However, it is guaranteed that exactly one process is privileged in each two subsequent PIF cycles. This is shown in the following theorem.

Theorem 5.7. For each legConfig γ_0, where $\gamma_0 \models \mathsf{root}_{\top}^{\top}$, for each execution $\gamma_0, \gamma_1, \ldots$, there exists $0 < i < j \leq 4 \cdot \mathsf{depth}(\mathcal{T})$, such that $\gamma_i \models \mathsf{root}_{\bot}^{\top}$, $\gamma_j \models \mathsf{root}_{\top}^{\top}$, and the action *critSection*() is executed by exactly one process within $\gamma_0, \ldots, \gamma_{4 \cdot \mathsf{depth}(\mathcal{T})}$.

Proof Sketch (the full proof is given in Appendix A.6). Let d be $\mathsf{depth}(\mathcal{T})$. By Theorem 5.6, any configuration following γ_0 is a legConfig. By Lemma 5.3, the root

switches into $\mathsf{root}^{\top}_{\bot}$ in $i \leq 2d$ steps. γ_i is either in *C2* or *C4*. From γ_i (cf. Figure 5.6), the root switches to $\mathsf{root}^{\top}_{\top}$ in $r \leq 2d$ steps, such that $\gamma_{i+r} \in$ *C1*. Now, if $\gamma_i \in$ *C2*, then there exists $i < e < r$, such that $\gamma_e \in$ *C3*, and exactly one non-root process is privileged in γ_{e-1}. Otherwise, $\gamma_i \in$ *C4*, and the root executes *critSection*() before reaching γ_r. The sum of i and r is less or equal to $4d$. □

Educated Unique Process Selection

The following lemma, the corollary, and the theorem concern the convergence time wrt. educated and unique process selection.

Lemma 5.4. Each legConfig satisfies educated unique process selection

Proof Sketch (the full proof is given in Appendix A.7). The properties of a legConfig satisfy the safety property of *EUPS*. For each execution starting from a legConfig (cf. Figure 5.6), each configuration is a legConfig, and the liveness property of *EUPS* holds in the execution (Theorem 5.7). *EUPS* holds in $3 \cdot \mathsf{depth}(\mathcal{T})$ steps by Theorem 5.5. □

Corollary 5.2. Given a configuration $\gamma_0 \in \Gamma$ of a topology, for each execution $\gamma_0, \gamma_1, \ldots$, educated unique process selection – *EUPS* – is satisfied in each configuration $\gamma_{3\cdot\mathsf{depth}(\mathcal{T})}, \gamma_{3\cdot\mathsf{depth}(\mathcal{T})+1}, \ldots$.

Proof. Follows from Lemma 5.4 and Theorem 5.5. □

Theorem 5.8. Given a configuration $\gamma_0 \in \Gamma$ of a topology, for each execution $\gamma_0, \gamma_1, \ldots$, unique process selection – *UPS* – is satisfied in each configuration $\gamma_{\mathsf{depth}(\mathcal{T})}, \gamma_{\mathsf{depth}(\mathcal{T})+1}, \ldots$.

Proof Sketch. Similar to proving Theorem 5.3 for Algorithm 5.5. □

5.7 Remarks

This section discusses some performance aspects of Algorithm 5.5 and Algorithm 5.6. It shows trade-offs between Algorithm 5.5 (resp. Algorithm 5.6) and other typical algorithms wrt. their desired properties, summarized in Table 5.1 (resp. Table 5.2). Again, the space complexity represents the size of the local state space for each process.

The typical algorithms considered in this section are of two sorts. First, the token-ring based algorithms are usually designed for ring topologies. The typical scenario of such algorithms is to pass a token in a circular manner through a ring. Such algorithms are applied to trees by circulating a token in a depth-first manner following an Euler cycle, and creating a virtual ring. Examples of such approaches are found in [Dol00, PV00, DJPV00, PV07]. In particular, the approach in [PV07] is snap-stabilizing to the property that when the root sends a token, the token returns to the root after visiting all processes in a depth-first manner. The approach in [PV07] is also space optimal: it requires $\mathsf{degree}(p)$ values for each process p ($\mathsf{degree}(p)$ is the number of children of p.)

The second sort is the unison or phase-clock based algorithms, explained in Section 4.3.1. Examples of such algorithms are the approaches of [BPV04, BPV08, DG13]

	Algorithm 5.5	Virtual Ring	Synchronous Unison
UPS-Convergence Time	$\mathsf{depth}(\mathcal{T})$	$\mathcal{O}(\mathsf{depth}(\mathcal{T}))$	$\lceil \mathsf{diam}(\mathcal{G})/2 \rceil - 1$
Recurrence Δ	$1/(4 \cdot \mathsf{depth}(\mathcal{T}))$	$1/(\mathsf{n} + \lvert\mathcal{E}\rvert - 1)$	$1/\mathsf{n}$
$priv_\Delta$-Convergence Time	$7 \cdot \mathsf{depth}(\mathcal{T})$	$\mathcal{O}(\mathsf{depth}(\mathcal{T}))$	$2 \cdot \mathsf{diam}(\mathcal{G}) + \mathsf{n}$
Space	$4(\mathsf{n} + 1)$	3	$\mathsf{n} + \mathsf{diam}(\mathcal{G})$

Table 5.1: Unique process selection of some sorts of algorithms, under the assumption that there is exactly one active process

	Algorithm 5.6	Virtual Ring
UPS-Convergence Time	$\mathsf{depth}(\mathcal{T})$	$\mathcal{O}(\mathsf{depth}(\mathcal{T}))$
$EUPS$-Convergence Time	$3 \cdot \mathsf{depth}(\mathcal{T})$	$\mathcal{O}(\mathsf{depth}(\mathcal{T}))$
Recurrence Δ	$1/(4 \cdot \mathsf{depth}(\mathcal{T}))$	$1/2(\mathsf{n} + \lvert\mathcal{E}\rvert - 1)$
Space	$4\mathsf{n}$	$3\mathsf{n}$

Table 5.2: Educated unique process selection of some sorts of algorithms

and Algorithms 4.1, 4.2, and 4.3 in Chapter 4. Recall that the condition *priv* is defined as follows: *priv* is satisfied by a configuration of a topology iff there exists one process that is privileged in the configuration.

Table 5.1 considers the property of unique process selection, under the assumption that *there always exists exactly one active process.* It shows the complexity results that can be achieved for each sort. Concerning the convergence time wrt. UPS, all algorithms converge in a time that is a factor of the tree depth or the graph diameter. The synchronous unison can achieve the best convergence time complexity. Concerning the recurrence that can be achieved, the virtual ring and the synchronous unison have a recurrence that is a reciprocal of n. However, in Algorithm 5.5, the recurrence is a reciprocal of the tree depth. Considering the $priv_\Delta$-convergence time, it is a factor of the tree depth or the graph diameter for all cases, similar to to the convergence wrt. UPS. Finally, the space requirement makes a difference among the tree approaches: it is most efficient in the virtual ring approach. To sum up, Algorithm 5.5 has a clear benefit over the other algorithms when the number of processes requesting a privilege is low, and when the tree depth is low compared to the number of processes. This efficiency lies in the high recurrence of *priv*, traded with the space requirement.

In Table 5.2, the property educated unique process selection is considered. In the related work, there exists no virtual ring approach for educated unique process selection. However, an approach can be designated as follows: a token is circulated through all processes twice. In the first circulation, the process with the highest value of m is marked. In the second circulation, a selected process is granted a privilege. For such an approach, lower bounds on the time, recurrence, and space complexities are given in Table 5.2. By observing the values of the virtual ring, the complexity gets worse from all aspects compared to the case of Table 5.1. However, the complexity for Algorithm 5.6 stays almost the same compared to Algorithm 5.5. Note that the $priv_\Delta$-convergence time is not considered here, and the reason is given in Section 5.6.2.

The time complexities given in Tables 5.1 and 5.2 are worst case complexities. From

an average case perspective, Algorithm 5.5 has a significance of having high recurrence of executing *critSection*(), due to having an immediate feedbacks after an active process is found or after an active process executes *critSection*(). The following is some average case analysis for Algorithm 5.5.

Let $\mathcal{T} = (\mathcal{P}, \mathcal{E})$ be a tree with a unique active process p. Let $\mathsf{avg_depth}(\mathcal{T})$ be the average depth for any process in $\mathcal{P}$. On average, Algorithm 5.5 requires $4 \cdot \mathsf{avg_depth}(\mathcal{T})$ steps to complete the two cycles, in which *critSection*() is executed. If the tree has a branching factor f, then $\mathsf{avg_depth}(\mathcal{T})$ is given as follows:

$$\mathsf{avg_depth}(\mathcal{T}) = \frac{\sum_{i=0}^{d} f^i \cdot i}{\mathsf{n}}.$$

Thus, the average number of steps of the two cycles is

$$4 \cdot \frac{\sum_{i=0}^{d} f^i \cdot i}{\mathsf{n}}.$$

For, e.g., virtual ring algorithms where a single token is passed in a depth-first manner creating a virtual ring, the number of steps to traverse a tree $\mathcal{T} = (\mathcal{P}, \mathcal{E})$ is $\mathsf{n} + |\mathcal{E}| - 1$. Thus, the average number of steps to reach the active process is $\frac{\mathsf{n}+|\mathcal{E}|-1}{2}$.

Figure 5.7 illustrates a comparison between Algorithm 5.5 and the virtual ring approach, for some trees with branching factors that range between 1–5, where there is only one active process. It is obvious that for trees with large branching factors and depth, Algorithm 5.5 is much more efficient than the virtual ring approach.

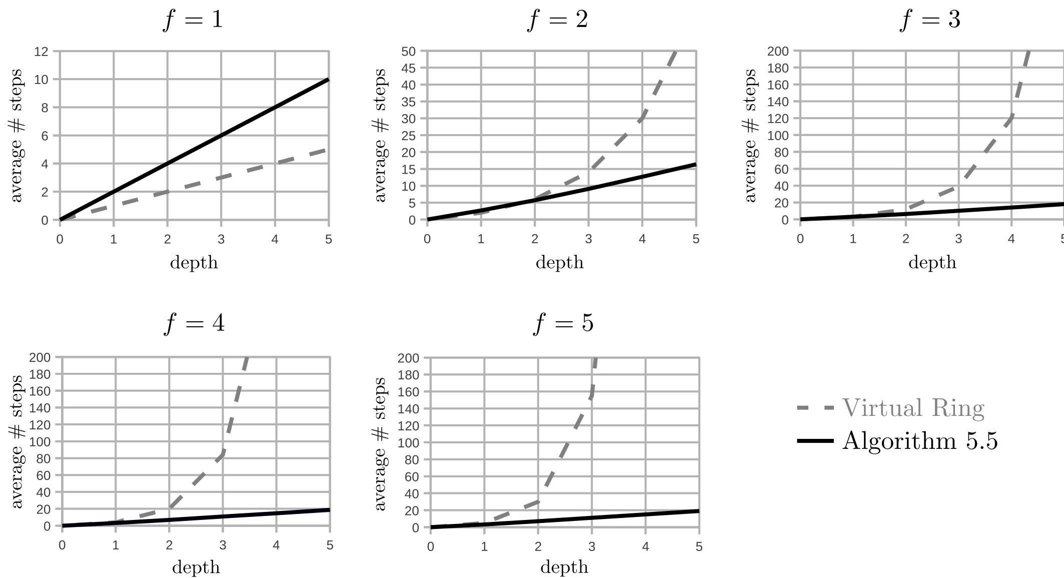


Figure 5.7: Performance of Algorithm 5.5 vs. the virtual ring approach

6 TDMA Slot Assignment

The problem of Time-Division-Multiple-Access (TDMA) slot assignment is considered in this chapter. In general, this problem concerns assigning nodes (i.e. components) of a network topology to time slots for the sake of scheduling communication between the nodes, when the communication medium allows only a limited number of messages between neighbors at each point in time. The aim in this chapter is to have slot assignment that is efficient wrt. clock synchronization, and the requirement of extra time intervals – namely the *guard time* – to avoid message collision or loss. The efficiency is optimized when the achieved slot assignment allows the most precise clock synchronization, such that the length of the required extra intervals is the least among all possible slot assignments. This entails having the highest recurrence of message delivery. Some of this chapter's content has appeared in [1, 2] of the author's publications.

The problem is concerned for wireless sensor networks that conform to the *European Norm* EN 54-25 [DIN05], which consists of the requirement specification of wireless fire alarm systems. The basic requirement of such a system is that it has to meet real-time deadlines for performing actions.

The structure of this chapter is as follows. Section 6.1 presents the TDMA slot assignment problem and its related work. Section 6.2 defines the problem statement and the contributions. Section 6.3 specifies the system model and some properties. Next, Section 6.4 presents analysis of the worst and best slot assignments wrt. clock synchronization. Section 6.5 provides a methodology for guard time optimization. Section 6.6 discusses recurrence properties and self-stabilization wrt. slot assignment for the given model. Finally, Section 6.7 presents a case study.

6.1 The TDMA Slot Assignment Problem

TDMA [Rap02] is designated for distributed systems, in which the communication medium allows only a limited number of messages between nodes. TDMA guarantees that the nodes share the communication medium without message collision or loss. This is achieved by scheduling the communication of the nodes over time. A general scheme of TDMA comprises dividing time into periodic intervals called *time frames*, where each frame is divided into *time slots*. Each slot is assigned to at most one node n among $\mathcal{N}_n^*$, and each node is allowed to send messages only during its assigned slots. This is, incidentally, some aspect of the local mutual exclusion problem [HP92]: if a process p is privileged, then for each $q \in \mathcal{N}_p$, q is not privileged. TDMA in particular provides an opportunity for saving energy by setting nodes into sleep – or power saving – mode during the time slots in which they do not send or listen. TDMA is used in some popular network communication protocols, such as TTP [KG93] and FlexRay [Par12].

As TDMA is based on time slots, nodes are supposed to be time synchronized. Otherwise, the system may exhibit two critical behaviors:

1. Two nearby nodes send messages at the same point in time. This might happen because each node supposes that the current time point points to the slot assigned to it for sending. This may cause message collision.

2. A node sends a message to a target node while the latter is in not listening, causing message loss.

Message collision or message loss may cause serious problems in time-critical systems: they may block, delay, or negatively alter the system reactivity.

Means to provide the same time to all network nodes depend on the network architecture. In the ideal case, all nodes share the same clock signal at the hardware level. Then, there is no need for additional clock synchronization mechanisms. However, this is not possible for all distributed systems, and even if possible, it might be too expensive. An alternative case is having one local hardware clock for each node. In such a case, temperature, voltage change, noise, etc. [PK00] lead to a variation in the crystal frequencies and thus *clock drift* may be experienced. For such cases, clock synchronization mechanisms are required. In case of, e.g., FlexRay [Par12], each node is equipped with a local clock, but all nodes are connected by a bus and receive a common synchronization signal at the beginning of each frame. However, in wireless sensor networks, it is not always the case that nodes have access to a common synchronization signal. For that, clock synchronization mechanisms – that may involve communication and timestamp exchange between multiple nodes – are needed.

Unfortunately, synchronization mechanisms cannot guarantee absolute precision for two reasons:

1. If the gap between two subsequent clock synchronizations of node is large, a clock may drift critically large in between.

2. A received clock value by some node may not be correct if the value is sent by a node that does not provide the reference clock.

A bounded clock drift can be tolerated by using the notion of *guard time* [PS13]. Guard time can be designed as two time intervals that are added to the beginning and the end of each slot. During the guard time of a slot, the assigned node for sending is not allowed to send messages, and the assigned node for listening is supposed to listen. With this setting, a bounded clock drift does not yield message collision or loss.

Guard time is considered as an addition to the slot length, which consequently extends the frame length. This reduces the performance of the system by two issues:

1. The time required to deliver messages through a path increases due to the increase of the frame length.

2. By extending the listening duration for each node, energy consumption increases.

From this point, guard time minimization is important. The classical-engineering approaches to guard time minimization in a system comprise extensive testing, simulation, and calculations based on experience with the system architecture, its requirements, and environmental conditions. Such approaches usually result in fair approximations of the optimal guard time length, and certain safety margins are added

to compensate for the incompleteness of these approaches. In addition, these approaches take a long time, require many system tests to be performed, and an effort that may be necessary for each modification of the system during further development.

The minimum guard time length required by a system over a topology is impacted by the slot assignment in some systems (cf. Section 6.4). This chapter concerns guard time minimization for networks conforming to EN 54-25 [DIN05]. Guard time minimization, in turn, increases the recurrence of sending messages.

6.1.1 Related Work

In literature, the TDMA slot assignment and clock synchronization problems are considered separately in most cases. This entails that the solutions to these problems for one system are built on several layers, especially, when it concerns self-stabilization.

An efficient TDMA slot assignment is being evaluated by many criteria: how fast the assignment is achieved, how fast the achieved message delivery is, how many slots per frame are occupied, how much the slot length is, and others.

In general, the TDMA slot assignment problem is mapped to the NP-complete *minimum graph coloring* problem [Gav72, Kar72, GK93], which investigates what is the minimum number of colors required to color nodes in a graph, such that no color is repeating within some distance. In relation to slot assignment, the colors represent slots, which achieves that: if a node is assigned to a slot, no other node within some distance may be assigned to the same slot, in order to avoid collision. A challenge to achieve this is to have a diversity of colors within each distance instead of having one color repeating, because diversity entails sharing the bandwidth efficiently.

The first TDMA slot assignment in the scope of self-stabilization with its typical models appears in [KA03a, KA03b]: a system stabilizes when all nodes are assigned to slots correctly. In [KA03a], the authors present algorithms that work for general graphs, but are suitable or efficient for two-dimensional grid topologies. The topologies use a simple message-passing model, with the following assumptions: the clocks are synchronized, they do not drift, and the time of message delivery between two nodes is constant. In [KA03b], the authors introduce the *write all with collision* (WAC) model. This model suits the communication nature of wireless sensor networks basically in two aspects:

1. In each step, a process changes its state and the public variables of its neighbors. This reflects broadcasting a message to the neighbors.

2. If two processes tend to change the state of some mutual neighbor q, then q's state remains the same. This models collision or message loss.

The work of [KA03b] shows transformational approaches from this model to the read and write model and vice versa, preserving self-stabilization.

Next, TDMA slot assignment is considered in [HT04] under the assumption that clocks are synchronized and the number of neighbors is upper-bounded. The approach tackles basically scalability issues given node failures and dynamic topology changes. Next, the work of [AK05] presents a self-stabilizing deterministic TDMA slot assignment. This approach comprises three levels. First, a self-stabilizing slot assignment for the

shared memory model is introduced. This algorithm is actually a minimum graph coloring algorithm. Second, this model is transformed to the WAC model, however, without preserving self-stabilization. Third, self-stabilization is added. Finally, the work of [PST14] provides a slot assignment algorithm, given some relaxed assumptions: there is no collision detection, no prior clock synchronization, and no external time reference, e.g. global pulse.

In regard to self-stabilization, what matters in this work is to analyze lower and upper bounds on the recurrence Δ of sending messages, that can be achieved after any system stabilizes. The self-stabilizing approaches mentioned above provide many insights and ideas on how to build a self-stabilizing slot assignment algorithm for this work. The approaches, however, may not be applied directly, since the model used in this chapter slightly differs from the prior models: in the new model, collision may happen between any two nodes in the tree, since structuring a tree may not necessarily depend on the physical location of nodes. Details are given in Section 6.3.

Clock synchronization for wireless sensor networks is considered intensively in literature. Clock synchronization methodologies can be categorized into four categories. The first category comprises *leader-based approaches*, in which clocks are synchronized based on a reference node. The Network Time Protocol (NTP) [Mil91] is an example that is used for networks with hierarchical structure, e.g. trees with a root representing reference time. In this protocol, messages sent through the hierarchy downwards are timestamped to propagate the clock value of the root to the nodes. The clocks are synchronized based on the root clock value. The second category reflects *Reference Broadcast Synchronization* (RBS) [EGE02], whose basic version has the following behavior: when two receivers receive a message, each of them records the time when it has received the message. Next, the two receivers exchange the recorded values and compute the offset. RBS is useful in networks where there is no source of correct time, or no reference node. The third category comprises *converge-to-max* protocols, in which timestamps are sent periodically, and a receiver adjusts its clock value by choosing the maximum one among its clock value and the received values. This protocol has a similar concept of the unison algorithms given in Chapter 4. An example of a self-stabilizing version of this protocol is [HZ06], in which clock drift is also considered. Finally, the *pulse-based* approaches [MS90, WTP+05] synchronize clocks by sending pulses that adjust biological oscillators. The network considered in this work has tree topology, where the communication nature between nodes facilitates using leader-based synchronization, efficiently, similar to the NTP protocol.

Guard time optimization, or minimization, is found in literature to be based on testing, simulation, and approximation based on experience. Examples are [OR08, HBTH14]. The results of these approaches are based on fixed systems, architectures, and assumptions. In contrast, this work provides a novel formal approach to guard time optimization for the given model.

Related work includes also other aspects of this topic that are not in the scope of this work. For example, the quality of slot assignment can be defined by the maximum possible number of assigned slots for each frame, e.g. [CP09]. Other approaches, as [WT06], focus on minimizing the slot length according to the message size. Concerning clock synchronization, some protocols – like the ALOHAnet protocols [Abr85] – consider the behavior of sensors, in sending, receiving messages and timestamps, within the slots.

6.2 Problem Statement and Assumptions

Recall that the network considered in this work is a wireless sensor network in the sense of EN 54-25 [DIN05]. Roughly, the network has tree topology, where each vertex – or node – in the tree is equipped with a hardware clock. The root is called a *central unit* providing the reference time, and the other nodes are called *sensors*. The nodes communicate with each other via radio signals over a shared frequency channel using TDMA. Each sensor sends a message to its parent within its slot, and the parent replies with an acknowledgment within the same slot. The acknowledgment is time-stamped with the parent's clock value, for clock synchronization.

Regarding message collision, this model differs from the WAC model: if a node sends a message, it is guaranteed that all neighbors receive the message, however, all nodes other than the neighbors may also receive the message (radio signal). In other words, collision may happen whenever any two nodes send messages simultaneously. This case is believed to be more realistic and safe than the WAC model, since a tree might be formed through a general graph without considering the physical location of nodes and the extra coverage of radio signals.

6.2.1 Problem Statement

The aim of this work is threefold. First, developing a formal model of the mentioned network and its properties. Second, finding the topological characterizations of the slot assignments that yield the most precise and the most imprecise clock synchronization, respectively. This impacts the requirement of the minimum length of safe – i.e. collision-free – guard time. Third, finding equations for computing the optimal guard time length for the given model, which facilitates computing the highest recurrence of sending messages.

Problem 6.1.

1. Provide a formal model for wireless sensor networks in the sense of EN 54-25 [DIN05].

2. Find the topological characterization of the slot assignments that require the smallest and the largest safe guard times, respectively, for the model.

3. Derive equations to compute the optimal guard time for each case.

4. Compute the greatest lower bound on the recurrence Δ of sending messages that is guaranteed for all slot assignments. Next, compute the least upper bound on the recurrence Δ of sending messages, that can be achieved by at least one slot assignment, under the restrictions of the model.

The contribution of this chapter is as follows:

- A formal model for wireless sensor networks in the sense of EN 54-25 [DIN05] – under some assumptions given in Section 6.2.2 – is presented.

- Based on the model and assumptions, the topological characterization of the slot assignments that require the largest and smallest guard times, respectively, are given. This specifies which slot assignments has the highest recurrence of sending messages.
- Equations to compute the optimal guard time for each of the previous cases are given.
- Tight lower and upper bounds on the recurrence Δ of sending messages for any slot assignment are given.
- A case study of a wireless fire alarm system is given. The aim of the case study is to show how the given model can be easily extended to match real-world situations.

6.2.2 Assumptions

Some restrictions are added to the model to simplify the analysis. However, if some realistic scenario violating the restrictions exists, then this is either due to a fault, or can be represented by a simple extension of the model. In the following, the general restrictions or assumptions on the model are given, together with their justification if it has not been mentioned before.

1. A *topology* is a tree $\mathcal{T} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of *nodes*. The root of $\mathcal{T}$ is called *central unit* and denoted by $\mathsf{cu}(\mathcal{T})$. Each node other than the root is called *sensor*. The set of sensors in $\mathcal{T}$ is denoted by $\mathsf{Sn}(\mathcal{T})$. Following the general assumption that $\mathsf{n} \geq 2$, and $\mathsf{depth}(\mathcal{T}) \geq 1$, it follows that $|\mathsf{Sn}(\mathcal{T})| \neq 0$ holds.
2. Within each frame, each node is assigned to exactly one slot per frame; i.e. a bijective assignment. This case seems to be ideal, and may not be easily achieved. However, the formal approach and results can be extended to cover an arbitrary number of slots within one frame.
3. The communication between a sensor and its parent is reliable, in a sense that each sensor is synchronized from its parent within its slot and once per frame. Situations violating this assumption are considered separately in the case study (cf. Section 6.7).
4. The assignment is fixed as long as there is no message collision or loss.
5. Initially, all clocks are synchronized, and the system is stable.

Note that regarding the second and third assumptions, if the employed synchronization mechanism uses the same communication medium that is managed by TDMA, then message loss or collision during a sensor's slot may inhibit proper synchronization in general; this assumption can be represented as requiring a separate reliable channel for synchronization. However, safe guard times have the property that message loss or collision is effectively avoided, and, thus, with a safe guard time, the synchronization mechanism may well use the shared medium managed by TDMA without being affected by message loss or collision.

6.3 System Model

This section introduces the system model. The model is based on powerful and rigorous mathematics with real-time aspects, which provides the possibility to formally analyze the problem.

6.3.1 Evolutions with Clock Drift

To define clock drift, message collision, and loss, it is sufficient to observe the clock value of each node in the topology, represented by a positive real number ($\mathbb{R}_0^+$), and whether a node is sending a message or listening, represented by two boolean ($\mathbb{B}$) values.

Definition 6.1 (Evolution)**.** An **evolution** over a topology $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ is an interpretation $\mathcal{I}$ of the variables $\mathsf{clk}_v : \mathsf{Time}\ (= \mathbb{R}_0^+)$, $\mathsf{send}_v : \mathbb{B}$, and $\mathsf{listen}_v : \mathbb{B}$ for $v \in \mathcal{V}$ such that

1. $\mathcal{I}(\mathsf{clk}_v)(0) = 0$ for each $v \in \mathcal{V}$, and
2. $\mathcal{I}(\mathsf{clk}_{\mathsf{cu}(\mathcal{T})})(t) = t$ for each $t \in \mathsf{Time}$.

$\mathsf{Evo}(\mathcal{T})$ is written to denote the set of all evolutions over $\mathcal{T}$. Furthermore, $\mathsf{clk}_v^{\mathcal{I}}(t)$, $\mathsf{send}_v^{\mathcal{I}}(t)$, and $\mathsf{listen}_v^{\mathcal{I}}(t)$ are written to denote $\mathcal{I}(\mathsf{clk}_v)(t)$, $\mathcal{I}(\mathsf{send}_v)(t)$, and $\mathcal{I}(\mathsf{listen}_v)(t)$, respectively. ♢

With Definition 6.1, the undesired conditions of message collision and loss can be precisely characterized. Recall that in this model, messages sent between a sensor and its parent do not collide with each other during the sensor's slot.

Definition 6.2 (Message Collision/Loss)**.** An evolution $\mathcal{I}$ over topology $\mathcal{T}$ is said to have

1. **message collision** at time $t \in \mathsf{Time}$ between two different sensors $v_1, v_2 \in \mathsf{Sn}(\mathcal{T})$ iff both send at t, i.e. if
$$\mathsf{send}_{v_1}^{\mathcal{I}}(t) \wedge \mathsf{send}_{v_2}^{\mathcal{I}}(t),$$
2. **message loss** at time $t \in \mathsf{Time}$ for sensor $v \in \mathsf{Sn}(\mathcal{T})$ iff v is sending at t while its parent is not listening, i.e. if
$$\mathsf{send}_v^{\mathcal{I}}(t) \wedge \neg \mathsf{listen}_{\mathsf{parent}(v)}^{\mathcal{I}}(t).$$

$\mathsf{coll}_{v_1,v_2}^{\mathcal{I}}(t)$ ($\mathsf{loss}_v^{\mathcal{I}}(t)$) is written iff $\mathcal{I}$ has a message collision (loss) at t between sensors v_1, v_2 (for sensor v). ♢

The *clock speed* at a node is, formally, the derivative of an evolution of clock values with respect to time. The *clock drift* is the difference between a clock and the reference clock. and the *drift rate* is the rate of change of the clock drift.

Note that the evolution of clock values is not restricted by Definition 6.1: clock values may arbitrarily change, in particular non-continuously. Thus, clock speed and drift rate are in general only partial functions.

Definition 6.3 (Clock Drift)**.** Let $\mathcal{I}$ be an evolution over a topology $\mathcal{T} = (\mathcal{V}, \mathcal{E})$.

1. The **clock speed** of node $v \in \mathcal{V}$ in $\mathcal{I}$, denoted by $\theta_v^{\mathcal{I}}$, is the first derivative of the interpretation of clk_v with respect to time, i.e.

$$\theta_v^{\mathcal{I}} = \frac{\partial}{\partial t} \, \mathsf{clk}_v^{\mathcal{I}}(t).$$

2. The **clock drift** of node $v \in \mathcal{V}$ in $\mathcal{I}$ at time $t \in \mathsf{Time}$, denoted by $\varrho_v^{\mathcal{I}}(t) \in \mathbb{R}$, is the difference between the clock values of v and the central unit at time t in $\mathcal{I}$, i.e.

$$\varrho_v^{\mathcal{I}}(t) = \mathsf{clk}_v^{\mathcal{I}}(t) - \mathsf{clk}_{\mathsf{cu}(\mathcal{T})}^{\mathcal{I}}(t).$$

3. The **drift rate** of node $v \in \mathcal{V}$ in $\mathcal{I}$, denoted by $\delta_v^{\mathcal{I}}$, is the first derivative of the clock drift of v in $\mathcal{I}$ with respect to time, i.e.

$$\delta_v^{\mathcal{I}} = \frac{\partial}{\partial t} \, \varrho_v^{\mathcal{I}}(t).$$

The superscript $\mathcal{I}$ may be omitted if the interpretation is clear from the context. ♢

6.3.2 Scheduled Communication

The notions of frame and slot employed by TDMA can be simply formalized as a partitioning of the time domain – cf. Figure 6.1.

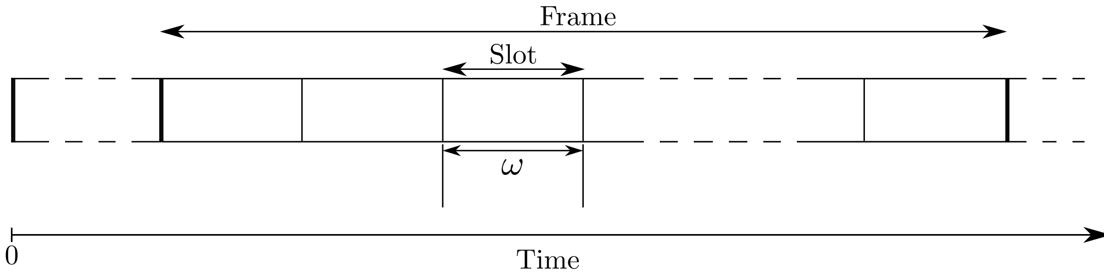


Figure 6.1: TDMA frames and slots

Definition 6.4 (Frame, Slot)**.** The **TDMA schedule** for a topology $\mathcal{T}$ with **slot length** $\omega \in \mathbb{R}^+$ and **number of slots per frame** $\mathsf{k} = |\mathsf{Sn}(\mathcal{T})|$ is a pair $(\mathsf{slot}, \mathsf{frm})$ of functions

$$\mathsf{frm} : \mathsf{Time} \to \mathbb{N}, \quad \text{and} \quad \mathsf{slot} : \mathsf{Time} \to \mathbb{N}$$

that are point-wise defined as

$$\mathsf{frm}(t) = \left\lfloor \frac{t}{\mathsf{k} \cdot \omega} \right\rfloor + 1, \quad \mathsf{slot}(t) = \left(\left\lfloor \frac{t}{\omega} \right\rfloor \bmod \mathsf{k} \right) + 1.$$

A time interval $[t_1, t_2)$ is called the *slot* (of $(\mathsf{slot}, \mathsf{frm})$) with *slot id* $(i, j) \in \mathbb{N} \times \mathbb{N}$ iff

$$t_2 - t_1 = \omega \wedge \forall t \in [t_1, t_2) \bullet \mathsf{slot}(t) = i \wedge \mathsf{frm}(t) = j. \qquad ♢$$

Recall that a slot assignment is just a mapping of sensors to slots in a bijective manner.

Definition 6.5 (Scheduled Evolution)**.** Let $\mathcal{T}$ be a topology having a TDMA schedule $(\mathsf{slot}, \mathsf{frm})$ with slot length ω and number of slots per frame $\mathsf{k} = |\mathsf{Sn}(\mathcal{T})|$. An evolution $\mathcal{I}$ over $\mathcal{T}$ is called **scheduled** wrt. $(\mathsf{slot}, \mathsf{frm})$ iff there exists an *assignment* of sensors to slots, i.e. a bijection

$$\mathsf{assign} : \mathsf{Sn}(\mathcal{T}) \rightarrow \{1, \dots, \mathsf{k}\}$$

such that:

1. Each sensor $v \in \mathsf{Sn}(\mathcal{T})$ sends messages only during the assigned slot according to its local clock, i.e.

$$\forall\, t \in \mathsf{Time} \bullet \mathsf{send}_v^{\mathcal{I}}(t) \longrightarrow \mathsf{slot}(\mathsf{clk}_v^{\mathcal{I}}(t)) = \mathsf{assign}(v).$$

2. For each sensor $v \in \mathsf{Sn}(\mathcal{T})$, its parent is listening in the slot assigned to v according to the parent's clock, i.e.

$$\forall\, t \in \mathsf{Time} \bullet \mathsf{slot}(\mathsf{clk}_{\mathsf{parent}(v)}^{\mathcal{I}}(t)) = \mathsf{assign}(v) \longrightarrow \mathsf{listen}_{\mathsf{parent}(v)}^{\mathcal{I}}(t).$$

The notation $\mathsf{Evo}(\mathcal{T}, \omega, \mathsf{assign}) \subseteq \mathsf{Evo}(\mathcal{T})$ denotes the set of evolutions scheduled by 'assign' for slot length ω. ♢

Figure 6.2 illustrates an example, where each frame has $|\mathsf{Sn}(\mathcal{T})|$ slots. Each slot is assigned to one of $\mathsf{Sn}(\mathcal{T})$, and the assignment is the same for all frames. This assignment is for sending; i.e. node v_1 sends a message only within the first slot of each frame.

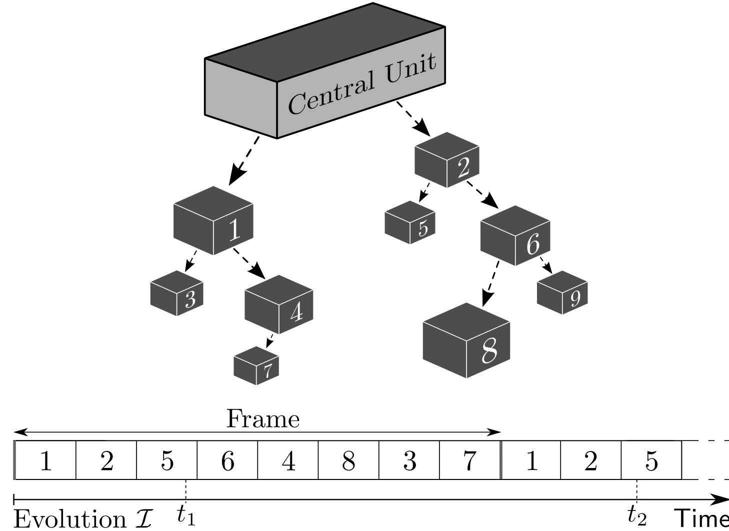


Figure 6.2: Scheduled evolution for sending (slot assignment)

6.3.3 Clock Synchronization

Clock synchronization is formalized by the notion of synchronized evolution in Definition 6.7 below. Two assumptions are taken into consideration. First, each sensor has exactly one synchronization point per frame. Second, the evolution of clock values is differentiable except for synchronization points. In order, for example, to model systems which use multiple synchronization points per slot, one can e.g. designate the latest synchronization points to be considered. In addition, as clock values evolve from synchronization points, it is assumed that the right-side derivative exists for all points in time. Furthermore, note that the notion of synchronized evolution models a wide range of explicit synchronization messages as well as timestamps.

Definition 6.6 (Synchronization Point)**.** Let $\mathcal{I}$ be an evolution over a topology $\mathcal{T}$. A point in time $t \in \mathsf{Time}$ is called **synchronization point** of sensor $v \in \mathsf{Sn}(\mathcal{T})$ in $\mathcal{I}$ iff the clock of v has the same value as the clock of its parent, i.e. if $\mathsf{clk}_v^{\mathcal{I}}(t) = \mathsf{clk}_{\mathsf{parent}(v)}^{\mathcal{I}}(t)$. ♢

Definition 6.7 (Synchronized Evolution)**.** Let $\mathcal{I}$ be an evolution over a topology $\mathcal{T}$ which is scheduled wrt. $(\mathsf{slot}, \mathsf{frm})$. $\mathcal{I}$ is called **synchronized** iff the following conditions hold:

1. Each sensor has at least one synchronization point in each of its slots, i.e.
$$\forall\, v \in \mathsf{Sn}(\mathcal{T}), j \in \mathbb{N}_0\ \exists\, t \in \mathsf{Time} \bullet \mathsf{frm}(t) = j \\ \wedge\, \mathsf{slot}(t) = \mathsf{assign}(v) \wedge \mathsf{clk}_v^{\mathcal{I}}(t) = \mathsf{clk}_{\mathsf{parent}(v)}^{\mathcal{I}}(t).$$
2. For each $v \in \mathcal{V}$, $\mathsf{clk}_v^{\mathcal{I}}$ is differentiable except for 0 and at most one point in each slot of v, i.e. if $\frac{\partial}{\partial t}\, \mathsf{clk}_v^{\mathcal{I}}(t)$ does not exist at $t, t' \in \mathsf{Time} \setminus \{0\}$ with $t \neq t'$, then there are two different slots $[t_1, t_2)$ and $[t_1', t_2')$ assigned to node v such that that $t \in [t_1, t_2)$ and $t' \in [t_1', t_2')$; the right-side derivative of $\mathsf{clk}_v^{\mathcal{I}}$ exists for all $t \in \mathsf{Time}$. ♢

In the example of Figure 6.2, the sensor v_5 is synchronized once within its assigned slot in each frame. This is achieved by copying the value of the clock of its parent (v_2). Given an evolution $\mathcal{I}$ with this assignment, v_5 may be synchronized in t_1 and t_2, which implies that: $\mathsf{clk}_{v_5}^{\mathcal{I}}(t_1) = \mathsf{clk}_{v_2}^{\mathcal{I}}(t_1)$ and $\mathsf{clk}_{v_5}^{\mathcal{I}}(t_2) = \mathsf{clk}_{v_2}^{\mathcal{I}}(t_2)$.

The following lemma and the notes concern properties of synchronization points and evolutions.

Lemma 6.1. Let $\mathcal{I}$ be a synchronized evolution over topology $\mathcal{T}$ with k slots per frame. The *distance* between two synchronization points of a sensor $v \in \mathsf{Sn}(\mathcal{T})$ is at most $(\mathsf{k}+1) \cdot \omega$, i.e.
$$\forall\, t \in \mathsf{Time}\ \exists\, t' \in \mathsf{Time} \bullet \\ 0 < t' - t < (\mathsf{k}+1) \cdot \omega \wedge \mathsf{clk}_v(t') = \mathsf{clk}_{\mathsf{parent}(v)}(t').$$

Proof. Let $v \in \mathsf{Sn}(\mathcal{T})$ be a sensor and $t \in \mathsf{Time}$. By Definition 6.7, there is a synchronization point in the next slot of v following t. In the worst case, t is the lower boundary of a slot of v and a synchronization point, then there is another synchronization point in the subsequent slot, at the upper boundary the latest. The claimed distance follows from Definitions 6.4 and 6.5. □

In phases where the evolution of a node's clock value is differentiable, there is the following relation between the current clock value, and an earlier clock value (of the node's parent) and the drift rate.

Lemma 6.2. Let $\mathcal{I}$ be an evolution over a topology $\mathcal{T}$ and let $\mathsf{clk}_v^{\mathcal{I}}$ be differentiable on the interval $(t_1, t_3) \subset \mathsf{Time}$.

1. Then
$$\forall\, t_2 \in [t_1, t_3) \bullet \varrho_v^{\mathcal{I}}(t_2) = \varrho_v^{\mathcal{I}}(t_1) + \int_{t_1}^{t_2} \delta_v^{\mathcal{I}}(t)\, dt.$$

2. If t_1 is a synchronization point of v, then
$$\forall\, t_2 \in [t_1, t_3) \bullet \varrho_v^{\mathcal{I}}(t_2) = \varrho_{\mathsf{parent}(v)}^{\mathcal{I}}(t_1) + \int_{t_1}^{t_2} \delta_v^{\mathcal{I}}(t)\, dt.$$

Proof. (1) It holds by Definition 6.3 and fundamental theorem of calculus. (2) It holds by Definition 6.6, Definition 6.3, and Point (1). □

Corollary 6.1. Let $\mathcal{I}$ be a synchronized evolution over a topology. It follows that $\theta_v^{\mathcal{I}}$ and $\delta_v^{\mathcal{I}}$ are defined on Time except for at most one point in each slot of v.

Proof. By Definition 6.7, $\mathsf{clk}_v^{\mathcal{I}}(t)$ is differentiable. □

6.3.4 Upper Bounds on Drift

The local clocks in such a system are typically determined by crystal devices. Usually, there are bounds on the quality of those devices and they are sensitive to environmental conditions such as temperature. Manufacturers of those devices often guarantee bounds on the drift rate for certain environmental conditions. In the following definition, an upper bound on the drift rate is specified.

Definition 6.8 (Bounded Drift Rate)**.** Let $\mathcal{I}$ be a synchronized evolution over topology $\mathcal{T}$. The value $\boldsymbol{\delta^{max}} \in \mathbb{R}_0^+$ is the *least upper bound* on the magnitude of the drift rate in $\mathcal{I}$ iff δ^{max} is the smallest number such that
$$\forall\, v \in \mathsf{Sn}(\mathcal{T}), t \in \mathsf{Time} \bullet |\delta_v^{\mathcal{I}}(t)| \leq \delta^{max}.$$

The notation $\mathbf{Evo}_{\boldsymbol{sync}}\boldsymbol{(\mathcal{T}, \omega, \mathsf{assign}, \delta^{max})}$ is used to denote the set of all synchronized evolutions over $\mathcal{T}$ with slot length ω which are scheduled by assign and for which $\delta^{max} \in \mathbb{R}_0^+$ is the least upper bound on the drift rate. ♢

In the following, the effect of the assignment on the maximum clock drift is studied in particular. The notion of maximum clock drift of an assignment in a given topology is defined.

Definition 6.9 (Maximum Clock Drift)**.** Let $\mathcal{T}$ be a topology. The **maximum clock drift** of node $v \in \mathsf{Sn}(\mathcal{T})$ under assignment 'assign' is called $\varrho_{\omega,\delta^{max}}^{max}(\mathsf{assign}, v) \in \mathbb{R}_0^+$ iff it

is the least upper bound on the clock drift of v in any synchronized evolution with slot length ω and least upper bound $\delta^{max} \in \mathbb{R}_0^+$ on the clock drift rates, i.e. if

$$\varrho^{max}_{\omega,\delta^{max}}(\mathsf{assign}, v) = \\ \mathsf{sup}\{\varrho_v^{\mathcal{I}}(t) \mid \mathcal{I} \in \mathsf{Evo}_{sync}(\mathcal{T}, \omega, \mathsf{assign}, \delta^{max}), t \in \mathsf{Time}\},$$

the *maximum clock drift* of assignment 'assign' in $\mathcal{T}$ is

$$\varrho^{max}_{\omega,\delta^{max}}(\mathsf{assign}) = \mathsf{sup}\{\varrho^{max}_{\omega,\delta^{max}}(\mathsf{assign}, v) \mid v \in \mathsf{Sn}(\mathcal{T})\}. \qquad \diamondsuit$$

6.4 Worst and Best Slot Assignments

This section concerns the worst and best slot assignments in terms of clock synchronization precision; i.e. in terms of the amount of clock drift that may be reached by clocks before they are synchronized. The worst assignment implies the highest possible value of maximum clock drift among all possible assignments. The best assignment yields the least possible value of maximum clock drift among all possible assignments.

The worst and best slot assignments are discussed in detail in the following sections. The worst and best assignments are roughly presented with an example sketched in Figure 6.3. It presents a topology of 9 sensors and a central unit.

- A slot assignment is a worst assignment iff there exists a path $p_0 = \mathsf{cu}(\mathcal{T}), \ldots, p_{d-1}, p_d$, such that (1) d is the tree depth, and (2) the sensors of the path are assigned reverse adjacent slots. An example of a worst assignment is given in Figure 6.3b.

- A slot assignment is a best assignment iff for each subtree rooted by a sensor at depth 1: (1) the subtree sensors are assigned adjacent slots. (2) each child is assigned a slot that is after the slot assigned to its parent. Figure 6.3c illustrates a best assignment.

To characterize the relevant differences between slot assignments, the notion of *forward distance* is introduced. The forward distance from a sensor v_1 to a sensor v_2 is simply the number of slots between any slot assigned to v_1 and the next slot assigned to v_2, which may lie in the same frame or in the subsequent frame. With this notion, the forward distances between sensors along a path can be summed up, to compute the time required to deliver a clock value from the first sensor to the last sensor in the path, given that each sensor is synchronized within its assigned slot. The maximum sum of forward distances of sensors along any path in the topology is denoted by $\mathcal{D}_{\mathsf{assign}}$. With $\mathcal{D}_{\mathsf{assign}}$, the maximum clock drift for any given assignment can be computed, and subsequently in Section 6.5, an optimal guard time can be computed as well.

Definition 6.10 (Forward Distance)**.** Given an assignment assign of slots to nodes for a topology $\mathcal{T}$, the **forward distance** between two sensors $v, v' \in \mathsf{Sn}(\mathcal{T})$ is defined by the function

$$\mathsf{fdist} : \mathsf{Sn}(\mathcal{T}) \times \mathsf{Sn}(\mathcal{T}) \longrightarrow \mathbb{N}$$

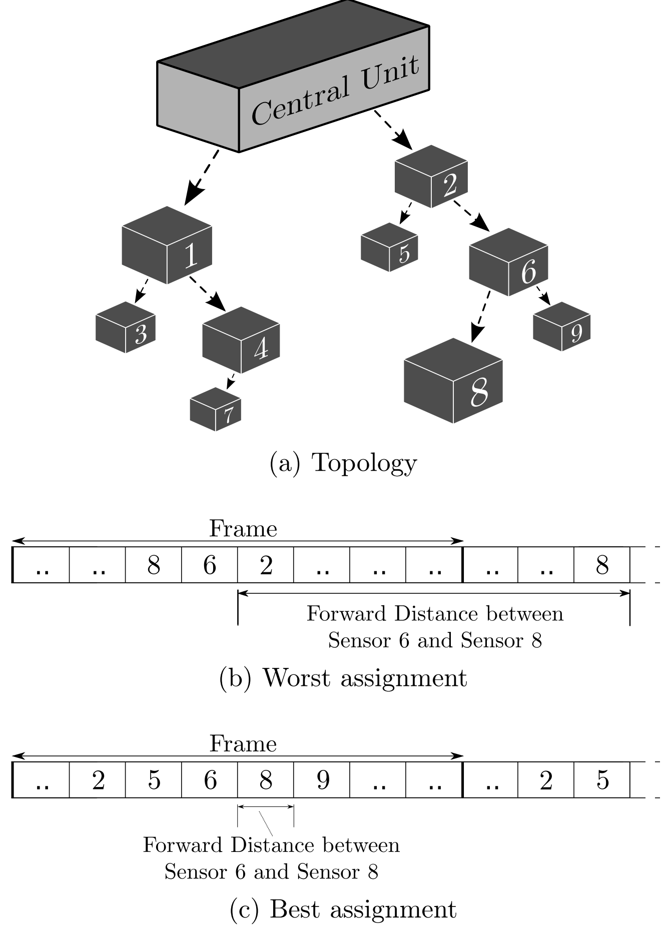


Figure 6.3: Example of worst and best assignments

which is defined point-wise as follows:

$$\mathsf{fdist}_{\mathsf{assign}}(v, v') = \begin{cases} \mathsf{assign}(v') - \mathsf{assign}(v) & \text{if } \mathsf{assign}(v') > \mathsf{assign}(v) \\ \mathsf{assign}(v') + |\mathsf{Sn}(\mathcal{T})| - \mathsf{assign}(v) & \text{if } \mathsf{assign}(v') \leq \mathsf{assign}(v). \end{cases}$$

The maximum of the sums of the forward distances between sensors on any path in $\mathcal{T}$ is denoted by $\mathcal{D}_{\mathsf{assign}}$:

$$\mathcal{D}_{\mathsf{assign}} = \max \left\{ \sum_{i=1}^{u-1} \mathsf{fdist}_{\mathsf{assign}}(v_i, v_{i+1}) \;\middle|\; v_1, \ldots, v_u \text{ path in } \mathcal{T}(v_1, \ldots, v_u \in \mathsf{Sn}(\mathcal{T})) \right\}.$$

$\diamondsuit$

The following lemma indicates how the maximum clock drift can be computed in reference to the maximum of the sums of forward distances.

Lemma 6.3. Let $v \in \mathsf{Sn}(\mathcal{T})$ be a sensor of a topology $\mathcal{T}$ of depth d. Given a TDMA schedule with k slots per frame:

$$\varrho^{max}_{\omega,\delta^{max}}(\mathsf{assign}, v) = \left(\sum_{i=1}^{d-1} \mathsf{fdist}_{\mathsf{assign}}(v_i, v_{i+1}) + \mathsf{k} + 1\right) \cdot \omega \cdot \delta^{max} \tag{6.1}$$

where $v_0, v_1, \ldots, v_d$ is the path from the central unit to node $v = v_d$ in $\mathcal{T}$.

Proof Sketch (the full proof is given in Appendix B.1). By induction over the depth of nodes. For the base case and the step, first show "$\leq$" using Lemma 6.2 and then "$\geq$" by construction. □

Corollary 6.2. Let $\mathcal{T}$ be a topology with TDMA schedule of k slots per frame, and let assign be an assignment of slots to nodes for $\mathcal{T}$. It follows that

$$\varrho^{max}_{\omega,\delta^{max}}(\mathsf{assign}) = (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1) \cdot \omega \cdot \delta^{max}.$$

Proof. It holds by Lemma 6.3 and Definition 6.10. □

Note that in the computation of the maximum clock drift, the number of slots per frame k is a parameter. In the given model, k is equal to the number of slots, due to the assumption of bijective assignments. However, Lemma 6.3 holds even for slot assignments that are not necessarily bijective, but guarantee that there is at least one assigned slot per frame for each sensor. This can be confirmed by a similar derivation to Lemma 6.3.

6.4.1 Worst Slot Assignment

The worst assignment reflects the assignment that yields the largest "sum of forward distances" along any path in the topology, which implies the largest clock drift. The largest sum of forward distances along a path is obtained as follows: let the number of slots in a frame be k. The forward distance from a sensor v_1 to a sensor v_2 can reach up to $\mathsf{k} - 1$ slots, only if any slot assigned to v_2 has a following slot assigned to v_1. For a path $v_1, \ldots, v_s$ with $s \in \mathbb{N}$ nodes, the largest sum of forward distances along the path can reach up to $(s-1)(\mathsf{k}-1)$ slots, which is maximized by choosing the largest s, namely the tree depth (cf. Figure 6.3b).

Lemma 6.4. Let assign be an assignment of slots to nodes for topology $\mathcal{T}$ of depth d with k slots per frame.

1. $\mathcal{D}_{\mathsf{assign}} \leq (d-1)(\mathsf{k}-1)$.
2. $\mathcal{D}_{\mathsf{assign}} = (d-1)(\mathsf{k}-1)$ iff there exists a path

   $$v_0, v_1, \ldots, v_d$$

   in $\mathcal{T}$ such that $v_0 = \mathsf{cu}(\mathcal{T})$, and the sensors on the path are assigned reverse adjacent slots by assign, i.e.

   $$\mathsf{assign}(v_i) = (\mathsf{assign}(v_{i+1}) + 1) \bmod \mathsf{k} \tag{6.2}$$

   for $1 \leq i < d$ (called (6.2)-path for short).

Proof Sketch (the full proof is given in Appendix B.2). For topologies of depth 1, both claims hold trivially, thus, let $\mathcal{T}$ be a topology of depth $d \geq 2$. Point (1) follows from the fact that assign has one assigned slot per frame for each sensor, and from Definition 6.10. For Point (2), show both directions of the bi-implication separately. □

6.4.2 Best Slot Assignment

The best assignment yields the least "maximum sum of forward distances" along any path in the topology, which implies the least value of "maximum clock drift experienced by any sensor" among all assignments. The maximum sum of forward distances along a path $v_1, \ldots, v_s$ is minimized when the forward distance between v_i and v_{i+1}, for $1 \leq i < s$, is minimized. The minimum possible forward distance between v_i and v_{i+1} is 1 when the slot assigned to v_{i+1} is the next slot adjacent to the slot assigned to v_i. However, this distance cannot be achieved for all paths; if v_i has another child v', and the forward distance from v_i to v_{i+1} is 1, then the distance from v_i to v' is greater than 1.

To minimize the maximum sum of forward distances along any path in the topology, the sensors belonging to each subtree have to be assigned adjacent slots, and within the adjacent slots, each child is assigned a slot that is after the slot assigned to its parent (cf. Figure 6.3c). Thus, in this case, surprisingly, the least possible maximum sum of forward distances along any path in the topology depends on the size of the largest subtree, not on the tree depth as such. In other words, the depth of a topology as such is not the limiting factor for clock precision.

Lemma 6.5. Let assign be an assignment of slots to nodes for topology $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ with k slots per frame and maximal subtree(s) of size $s \in \mathbb{N}$.

1. $\mathcal{D}_{\mathsf{assign}} \geq s - 1$.
2. $\mathcal{D}_{\mathsf{assign}} = s - 1$ iff
   a) For each path $v_0, v_1, \ldots, v_u$, where $v_0 = \mathsf{cu}(\mathcal{T})$, the forward distance between v_1 and each v_i, for $1 < i \leq u$, is at most $s - 1$, i.e.

   $$\forall\, 1 < i \leq u \bullet \mathsf{fdist}_{\mathsf{assign}}(v_1, v_i) \leq (s-1)$$

   Note that, given that the maximal tree size is s, the slots of any subtree of size s are adjacent by this condition.

   b) Each child is assigned a slot that is after the slot assigned to its parent, i.e.

   $$\forall\, v, v', v'' \in \mathsf{Sn}(\mathcal{T}) \bullet (v, v'), (v', v'') \in \mathcal{E} \\ \longrightarrow \mathsf{fdist}_{\mathsf{assign}}(v, v') < \mathsf{fdist}_{\mathsf{assign}}(v, v'')$$

Proof Sketch (the full proof is given in Appendix B.3). For topologies of depth 1, the two claims hold trivially. For topologies of depth greater than 1, first, it is shown that the conditions (2a) and (2b) establish forward distances which imply $\mathcal{D}_{\mathsf{assign}} = s-1$. Second, it is shown that if any of (2a) and (2b) is violated, then $\mathcal{D}_{\mathsf{assign}} > s - 1$ holds following Definition 6.10. □

In the example given in Figure 6.3, the worst assignment (Figure 6.3b) adheres to the conditions of Lemma 6.4, where the forward distance between parent and child is maximized. The best assignment (Figure 6.3c) adheres to Lemma 6.5, where the forward distance between parent and child is minimized

6.5 Guard Time Optimization

Guard time comprises two time subintervals at the beginning and the end of each time slot, respectively, for the sake of tolerating small clock drifts during synchronization: a sensor obeys a guard time $\phi \in \mathbb{R}_0^+$ if it does not send for a duration of ϕ at the beginning and the end of its assigned slot – cf. Figure 6.4. Note that obeying guard time is defined in terms of the local clock of the sensor: the sensor does not send if its local clock points to a value within the guard time of the sensor's slot. Note also that guard time is not necessarily a requirement in EN 54-25 [DIN05].

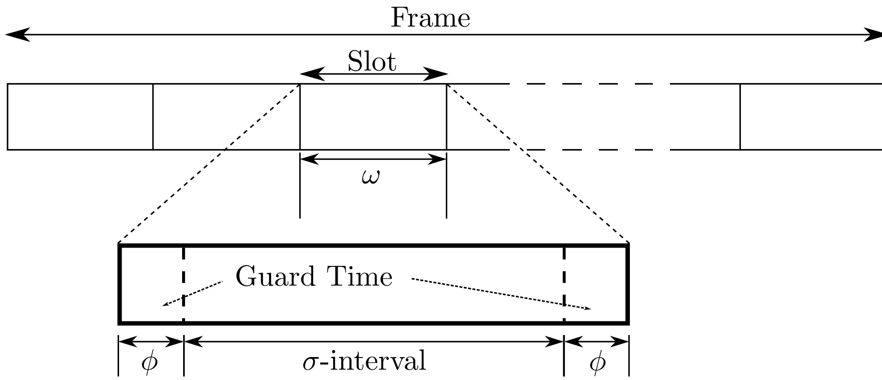


Figure 6.4: Guard time

The notion of guard time is formally defined in the following definition.

Definition 6.11 (Guard Time). An evolution $\mathcal{I}$ over topology $\mathcal{T}$ has **guard time** $\phi \in \mathbb{R}_0^+$ iff (1) $\mathcal{I}$ is scheduled with a slot length $\omega \geq 2\phi$ and (2) sensors do not send for a duration of ϕ at the beginning and the end of their slot, i.e.

$$\forall\, v \in \mathsf{Sn}(\mathcal{T}), t \in \mathsf{Time} \bullet \mathsf{send}_v^{\mathcal{I}}(t) \longrightarrow \\ \mathsf{slot}(\mathsf{clk}_v^{\mathcal{I}}(t) - \phi) = \mathsf{slot}(\mathsf{clk}_v^{\mathcal{I}}(t) + \phi) = \mathsf{slot}(\mathsf{clk}_v^{\mathcal{I}}(t)).$$

For each slot $[t_1, t_2)$ of $\mathcal{I}$ with guard time ϕ, the time intervals $[t_1, t_1 + \phi)$ and $[t_2 - \phi, t_2)$ are called the *(left and right) guard intervals* of the slot. The time interval $[t_1 + \phi, t_2 - \phi)$ is called *σ-interval* of the slot. ♢

6.5.1 Safe Guard Time

The length of the guard time is a critical issue for avoiding message collision and loss. The length should satisfy some conditions in order achieve its aim. A guard time, whose length is sufficient to avoid message collision and loss, is called *safe guard time.*

Definition 6.12 (Safe Guard Time)**.** A guard time $\phi \in \mathbb{R}_0^+$ is said to be **safe** for a topology $\mathcal{T}$, slot length ω, schedule assign, and least upper bound $\delta^{max} \in \mathbb{R}_0^+$ on the drift rates iff no synchronized evolution

$$\mathcal{I} \in \mathsf{Evo}_{sync}(\mathcal{T}, \omega, \mathsf{assign}, \delta^{max})$$

exhibits message collision or message loss. ♢

The following theorem derives, given a guard time ϕ, sufficient conditions on the guard time length in relation to clock drift in order to effectively avoid message collision and loss; i.e. to satisfy that the guard time is safe. The theorem states two facts. First, message collision does not happen if the absolute value of the clock drift experienced by any sensor at any time point does not exceed the value of ϕ. Second, message loss does not happen if the absolute value of the clock drift experienced by any sensor at any time point does not exceed the value of $\frac{\phi}{2}$.

Theorem 6.1. Let $\mathcal{I}$ be a scheduled evolution over topology $\mathcal{T}$ with guard time ϕ.

1. $\mathcal{I}$ does not have any message collision if

$$\forall\, v \in \mathsf{Sn}(\mathcal{T}), t \in \mathsf{Time} \bullet |\varrho_v^{\mathcal{I}}(t)| \leq \phi.$$

2. $\mathcal{I}$ does not have any message loss if

$$\forall\, v \in \mathsf{Sn}(\mathcal{T}), t \in \mathsf{Time} \bullet |\varrho_v^{\mathcal{I}}(t)| \leq \frac{\phi}{2}.$$

Proof Sketch (the full proof is given in Appendix B.4).

1. By premise, sensors obey the guard time and, because clock drift is bounded by the guard time, they send in their slot wrt. the central unit clock. Any message collision would yield a contradiction to that each sensor is assigned to one slot.

2. By premise, sensors obey the guard time and, because clock drift is bounded by $\frac{\phi}{2}$, they send well inside their slots wrt. the central unit clock. Parents listen throughout the slots of their children. This phase comprises, by the bound on clock drift, all sending points of the children, so there is no message loss. □

Corollary 6.3. A guard time $\phi \in \mathbb{R}_0^+$ is *safe* for a topology $\mathcal{T}$, slot length ω, schedule assign, and least upper bound $\delta^{max} \in \mathbb{R}_0^+$ on the drift rates iff

$$\varrho_{\omega,\delta^{max}}^{max}(\mathsf{assign}) \leq \frac{\phi}{2}.$$

Proof. It follows from Theorem 6.1 and Definition 6.12. □

To illustrate Theorem 6.1 and Corollary 6.3, consider the example in Figure 6.5. It assumes a synchronized system over a tree topology $\mathcal{T}$ with four sensors $v_a, v_b, v_c, v_d \in \mathcal{V}$ where v_c is parent of v_a and v_d is parent of v_b. Slots a and b are assigned to v_a and v_b, respectively, for sending. Given a guard time ϕ, different evolutions with different clock drift values for the given sensors and the possibility of message collision or loss are shown. Note that by Definition 6.5, parents v_c and v_d listen during slots a and b, respectively.

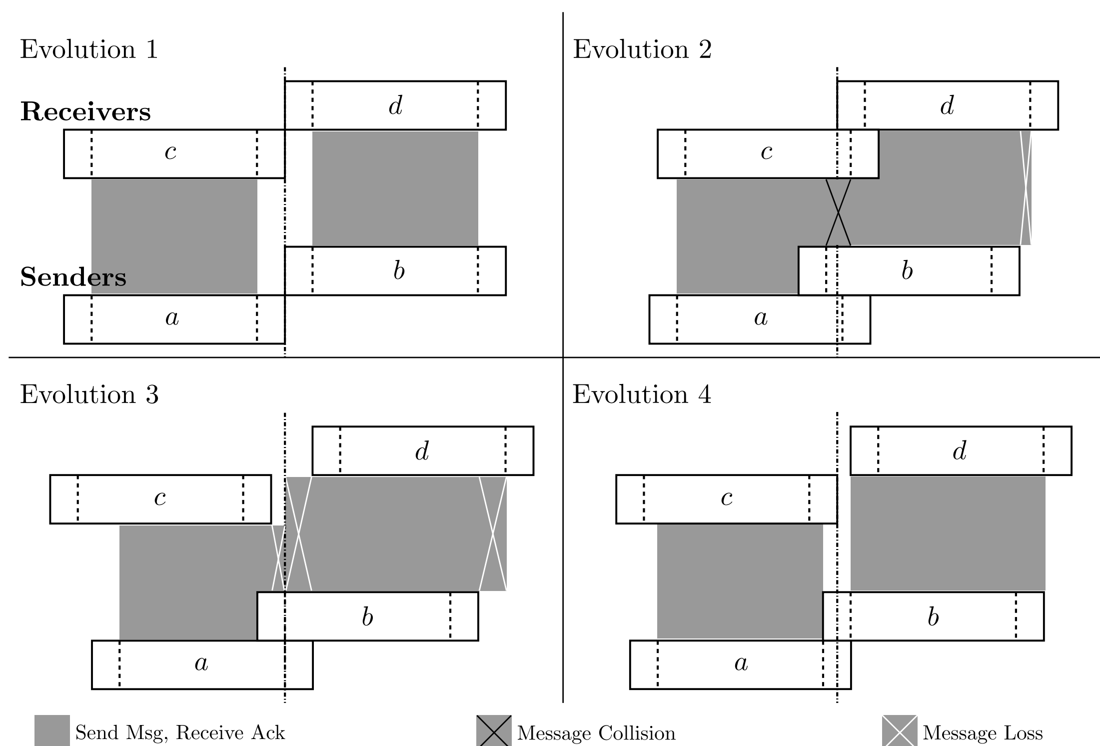


Figure 6.5: Safe guard time

- In Evolution 1, all clocks run at the same speed, and therefore there is no message collision or loss.
- In Evolution 2, there is clock drift with an absolute value that is greater than ϕ. Message collision between v_a, v_b, and message loss for v_b occur.
- In Evolution 3, the absolute values of clock drift do not exceed ϕ but only $\frac{\phi}{2}$. There is no message collision, but there exists message loss for at least v_b.
- In Evolution 4, the absolute values of clock drift do not exceed $\frac{\phi}{2}$, thus neither message collision nor loss are exhibited.

In general, the absence of collision and loss does not necessarily imply a violation of these conditions. For example, theoretically, a sensor clock may have a high speed in the first half of a frame and a low speed in the second half of the frame in a way such that it has a big maximal clock drift but still is perfectly on time for its next slot.

In the following lemma, sufficient conditions for message collision and loss are observed for completeness. It states that the bounds given by Theorem 6.1 are optimal, which means that evolutions with guard time strictly smaller than ϕ (or $\frac{\phi}{2}$) may have message collision (or loss).

Lemma 6.6. Let $\mathcal{I}$ be a scheduled evolution over topology $\mathcal{T}$ with slot length ω.

1. If there are two sensors $v_1, v_2 \in \mathsf{Sn}(\mathcal{T})$ and a point in time $t \in \mathsf{Time}$ such that
   - $\mathsf{assign}(v_2) = \mathsf{assign}(v_1) + 1$,
   - $t = t_2$ for a slot $[t_1, t_2)$ of v_1,
   - both nodes send continuously during their σ-interval, and
   - $\varrho^{\mathcal{I}}_{v_1}(t) = \phi + \tau$ and $\varrho^{\mathcal{I}}_{v_2}(t) = -\phi$ for $0 < \tau < \omega$,

   then there is message collision between v_1 and v_2 at t.
2. If there is a sensor $v \in \mathsf{Sn}(\mathcal{T})$ and a point in time $t \in \mathsf{Time}$ such that
   - $t = t_2 - \frac{\phi}{2}$ for a slot $[t_1, t_2)$ of v,
   - v sends continuously during its σ-interval, and
   - $\varrho^{\mathcal{I}}_{v}(t) = \frac{\phi}{2} + \tau$ and $\varrho^{\mathcal{I}}_{\mathsf{parent}(v)}(t) = -\frac{\phi}{2}$ for $0 < \tau < \omega$,

   then there is message loss at t.

Proof Sketch (the full proof is given in Appendix B.5). By construction. There exists an evolution which satisfies the premises and, by sending throughout the σ-interval, exhibit collision between nodes v_1 and v_2 and loss of a message from v, respectively. □

Note that a safe guard time need not exist for a given topology, slot length, assignment, and clock drift bound. If local clocks drift with a large amount during a frame, collision or loss may happen before a re-synchronization of clocks is possible.

6.5.2 Formal Derivation of Optimal Guard Time

The derivation – or computation – of an optimal guard time can be achieved, given basically a schedule, using the notion of the forward distance. Theorem 6.2 characterizes the existence and value of an optimal safe guard time basically in terms of $\mathcal{D}_{\mathsf{assign}}$; i.e. in terms of the maximum sum of forward distances of sensors along any path in the topology.

Theorem 6.2. Let $\mathcal{T}$ be a topology with k sensors and let $\sigma \in \mathbb{R}^+$ be the length of the σ-intervals. There exists an optimal, i.e. smallest, safe guard time for $\mathcal{T}$ wrt. the least upper bound $\delta^{max} \in \mathbb{R}^+_0$ on the clock drift rates iff

$$\delta^{max} < \frac{1}{4 \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1)}.$$

The optimal safe guard time for $\mathcal{T}$ wrt. δ^{max} is given by

$$\phi_{opt} = \sigma \cdot \frac{2 \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1) \cdot \delta^{max}}{1 - 4 \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1) \cdot \delta^{max}}.$$

Proof Sketch (the full proof is given in Appendix B.6). It follows the solutions of the equation system induced by the following: (1) a sufficient and necessary criterion for $\phi \in \mathsf{Time}$ being a safe guard time provided by Theorem 6.1 and Lemma 6.6, and (2) the value of $\varrho^{max}_{\omega,\delta^{max}}(\mathsf{assign})$ given by Corollary 6.2. □

By Definition 6.10, the forward distances and, thus, the maximum sum $\mathcal{D}_{\mathsf{assign}}$ depend on the slot assignment. In the following parts, the results indicated by Theorem 6.2 are translated to each of the worst and best case assignments.

Optimal Guard Time for the Worst Assignment

An optimal guard time for the worst assignment is safe for any other slot assignment, since $\mathcal{D}_{\mathsf{assign}}$ is the maximum for such an assignment. The derivation approach to optimal guard time for this case follows directly from Lemma 6.3, Lemma 6.4, and Theorem 6.1.

Corollary 6.4. Let $\mathcal{T}$ be a topology of depth d with k slots per frame. Let $\delta^{max} \in \mathbb{R}_0^+$ be a least upper bound on the clock drift rates, $\omega \in \mathbb{R}^+$ a slot length, and 'assign' an assignment of nodes to slots.

1. $\varrho^{max}_{\omega,\delta^{max}}(\mathsf{assign}) \leq (d(\mathsf{k}-1)+2) \cdot \omega \cdot \delta^{max}$.
2. $\varrho^{max}_{\omega,\delta^{max}}(\mathsf{assign}) = (d(\mathsf{k}-1)+2) \cdot \omega \cdot \delta^{max}$ iff 'assign' has a (6.2)-path.
3. Let $\sigma \in \mathbb{R}^+$ be the length of the σ-intervals. There exists a safe guard time for $\mathcal{T}$ wrt. δ^{max} iff
$$\delta^{max} < \frac{1}{4(d(\mathsf{k}-1)+2)}.$$
A safe guard time for $\mathcal{T}$ wrt. δ^{max} is given by
$$\phi_{opt} = \sigma \cdot \frac{2(d(\mathsf{k}-1)+2) \cdot \delta^{max}}{1-4(d(\mathsf{k}-1)+2) \cdot \delta^{max}}.$$
For assignments assign with a (6.2)-path, ϕ_{opt} is the optimal, i.e. smallest safe guard time.

Proof. Points (1) and (2) follow from Lemma 6.4,
$$(d-1)(\mathsf{k}-1)+\mathsf{k}+1 = d(\mathsf{k}-1)+2, \tag{6.3}$$
and Lemma 6.3. Point (3) follows from Points (1) and (2), and Lemma 6.4, Equation (6.3), and Theorem 6.2. □

Optimal Guard Time for the Best Assignment

An optimal guard time for the best case assignment is the minimum guard that is safe for at least one assignment. The derivation of such a guard time follows directly from Lemma 6.3, Lemma 6.5, and Theorem 6.2.

Corollary 6.5. Let $\mathcal{T}$ be a topology of with k slots per frame and maximal subtree(s) of size $s \in \mathbb{N}$. Let $\delta^{max} \in \mathbb{R}_0^+$ be a least upper bound on the clock drift rates, $\omega \in \mathbb{R}^+$ a slot length, and assign an assignment of nodes to slots.

1. $\varrho^{max}_{\omega,\delta^{max}}(\mathsf{assign}) \geq (s+\mathsf{k}) \cdot \omega \cdot \delta^{max}$.
2. $\varrho^{max}_{\omega,\delta^{max}}(\mathsf{assign}) = (s+\mathsf{k}) \cdot \omega \cdot \delta^{max}$ iff assign satisfies the conditions of Lemma 6.5–2b.

3. Let $\sigma \in \mathbb{R}^+$ be the length of the σ-intervals. There exists a safe guard time for $\mathcal{T}$ wrt. δ^{max} iff
$$\delta^{max} < \frac{1}{4(s + \mathsf{k})}.$$
The optimal safe guard time for $\mathcal{T}$ wrt. δ^{max} is given by
$$\phi_{opt} = \sigma \cdot \frac{2(s + \mathsf{k}) \cdot \delta^{max}}{1 - 4(s + \mathsf{k}) \cdot \delta^{max}}.$$
It is safe for exactly those assignments of nodes to slots which satisfy the conditions of Lemma 6.5.

Proof. Points (1) and (2) follow from Lemma 6.3, Lemma 6.5, and
$$(s - 1) + \mathsf{k} + 1 = s + \mathsf{k}. \tag{6.4}$$
Point (3) follows from points (1) and (2), and Lemma 6.5, equation (6.4), and Theorem 6.2. □

Note that for topologies of depth 1, the optimal guard time does not depend on the assignment, as the following Corollary indicates.

Corollary 6.6. Let $\mathcal{T}$ be a topology of depth 1 with k slots per frame. Let $\delta^{max} \in \mathbb{R}_0^+$ be a least upper bound on the clock drift rates, $\omega \in \mathbb{R}^+$ a slot length, and assign an assignment of nodes to slots.

1. $\varrho^{max}_{\omega,\delta^{max}}(\mathsf{assign}) = (\mathsf{k} + 1) \cdot \omega \cdot \delta^{max}$.

2. Let $\sigma \in \mathbb{R}^+$ be the length of the σ-intervals. There exists a safe guard time for $\mathcal{T}$ wrt. δ^{max} iff
$$\delta^{max} < \frac{1}{4(\mathsf{k} + 1)},$$
the optimal safe guard time for $\mathcal{T}$ wrt. δ^{max} is given by
$$\phi_{opt} = \sigma \cdot \frac{2(\mathsf{k} + 1) \cdot \delta^{max}}{1 - 4(\mathsf{k} + 1) \cdot \delta^{max}}.$$

Proof. Corollaries 6.4 and 6.5. □

6.6 Recurrence Property

The results of Corollaries 6.4 and 6.5 provide as well a method for computing the achieved recurrence of intervals in which collision- and loss-free messages are allowed to be sent, namely the σ-intervals. Indeed, the results provide upper and lower bounds on the recurrence that can be achieved by any self-stabilizing algorithm wrt. scheduled, synchronized, collision- and loss-free evolutions.

In the system model, the clocks may drift arbitrarily within some bounds. This fact does not hinder computing optimal guard times for collision- and loss-free

communication, as shown in the previous sections. However, concerning recurrence properties, this model may accept unrealistic behaviors, whose concern provides useless results. For example, the model accepts that a clock is too fast during σ-intervals and too slow during the guard time. This behavior practically entails low recurrence of σ-intervals. The other way around induces high recurrence. In practice, clock speed does not change that quickly.

Therefore, the analysis of recurrence concerns the partitioning of time intervals, regardless of clock speeds. In other words, the recurrence is analyzed according to reference time, namely the central unit.

Concerning self-stabilizing algorithms wrt. slot assignment, when collision or loss is detected, the approaches are usually based on setting nodes into listening mode, until slot assignment is re-established and clocks are synchronized, regardless of the guard time length (cf. Section 6.1.1). However, according to the analysis above, guard time is used to guarantee collision- and loss-free evolutions. Therefore, in the following parts, it is assumed that *any self-stabilizing algorithm wrt. scheduled and synchronized evolutions applies to any safe guard time.*

Recall from Section 3.3 that for real-time systems, the condition concerned with the recurrence property is defined over time points. Roughly, the condition for this case is satisfied at some point in time if there is a sensor in the σ-interval of its assigned clock according to the central unit's clock. To define this formally, the following notation is added: given a time point t and a sensor v, the predicate $\mathsf{slot}^v_\sigma(t)$ holds iff t is in the σ-interval of a slot assigned to v.

The condition concerned with the recurrence property is denoted by *safeSend*, and is defined as follows: for each evolution $\mathcal{I}$ and each point in time t:

$$\mathit{safeSend}(t) \longrightarrow \exists\, v \in \mathcal{V} \bullet \mathsf{slot}^v_\sigma(t).$$

The recurrence property to be concerned is $\mathit{safeSend}_\Delta$, where Δ is determined based on the slot assignments. The following theorem states a lower bound on Δ, such that the $\mathit{safeSend}_\Delta$ holds for any self-stabilizing algorithm wrt. scheduled, synchronized, and collision- and loss-free evolutions with safe guard time. It is inspired from the fact that the optimal guard time for the worst case is safe for all assignments.

Theorem 6.3. Given a topology with k slots per frame and depth d, let $\sigma \in \mathbb{R}^+$ be the length of the σ-intervals, and let δ^{max} be a least upper bound on the drift rate. If $\delta^{max} < \frac{1}{4(d(\mathsf{k}-1)+2)}$, then each self-stabilizing algorithm wrt. scheduled, synchronized, and collision- and loss-free evolutions with safe guard time is also a self-stabilizing algorithm wrt. $\mathit{safeSend}_\Delta$, where

$$\Delta = 1 - 4(d(\mathsf{k}-1)+2) \cdot \delta^{max}.$$

Proof. Let A be an algorithm that is self-stabilizing wrt. scheduled, synchronized, and collision- and loss-free evolutions given some safe guard time. By Corollary 6.4, Lemma 6.4, and assumption, each evolution with any assignment is finally collision- and loss-free if the optimal guard time for worst assignments is used (cf. Figure 6.3b for illustration about the worst assignment.)

Consider a synchronized and scheduled evolution $\mathcal{I}$ that starts at a point in time t that is the start point of an σ-interval, where any following point is collision- and

loss-free. By Definition 6.11, for each $t' \geq t$, the recurrence of *safeSend* in $[t, t']$ for $\mathcal{I}$ – $\mathsf{Rec}_{safeSend}(\mathcal{I}[t,t'])$ – is greater than or equal to $\frac{\sigma}{\sigma+2\phi}$. This, by Definition 3.6 (Definition of con_Δ in Section 3.3), implies that $safeSend_\Delta$ holds if $\Delta = \frac{\sigma}{\sigma+2\phi}$. Among all possible assignments, the minimum value of $\frac{\sigma}{\sigma+2\phi}$ is achieved when the value of ϕ is maximized; i.e. when the optimal guard time for a worst assignment is employed. By substituting the value of the optimal guard time for the worst assignment (Corollary 6.4) with ϕ, $safeSend_\Delta$ holds if $\Delta = 1 - 4(d(\mathsf{k}-1)+2) \cdot \delta^{max}$ holds.

This implies that A is self-stabilizing wrt. $safeSend_\Delta$ if $\Delta = 1-4(d(\mathsf{k}-1)+2)\cdot\delta^{max}$. □

The next theorem shows an upper bound on the achieved recurrence by any self-stabilizing algorithm wrt. scheduled, synchronized, and collision- and loss-free evolutions with safe guard time, inspired from the case of the best assignments.

Theorem 6.4. Let $\mathcal{T}$ be a topology with k slots per frame and maximal subtree of size s. Let $\sigma \in \mathbb{R}^+$ be the length of the σ-intervals, and let δ^{max} be a least upper bound on the drift rate. For each self-stabilizing algorithm wrt. scheduled, synchronized, and collision- and loss-free evolutions with safe guard time and wrt. $safeSend_\Delta$ for some $\Delta \in [0,1] \subset \mathbb{R}_0^+$, the following statement holds:

$$\Delta \leq 1 - 4(s+\mathsf{k}) \cdot \delta^{max}.$$

Proof. Let A be an algorithm that is self-stabilizing wrt. scheduled, synchronized, and collision- and loss-free evolutions with safe guard time and wrt. $safeSend_\Delta$, for some $\Delta \in [0,1] \subset \mathbb{R}_0^+$. This implies that for each evolution $\mathcal{I}$, there exists a time point $t \in \mathsf{Time}$ such that for all $t' \geq t$, $\mathsf{Rec}_{safeSend}(\mathcal{I}[t,t']) \geq \Delta$. Assume by contradiction that $\Delta > 1 - 4(s+\mathsf{k}) \cdot \delta^{max}$, which then implies that

$$\mathsf{Rec}_{safeSend}(\mathcal{I}[t,t']) > 1 - 4(s+\mathsf{k}) \cdot \delta^{max}. \tag{6.5}$$

By Definitions 6.4 and 6.11 (time division and guard time) and the definition of *safeSend*, there exists a time point $t'' \geq t$ such that

$$\mathsf{Rec}_{safeSend}(\mathcal{I}[t,t'']) = \frac{\sigma}{\sigma+2\phi}. \tag{6.6}$$

Thus, since σ is constant, the maximum possible value of $\mathsf{Rec}_{safeSend}(\mathcal{I}[t,t''])$ (with safe guard time) is given by having the minimum possible value of safe guard time ϕ. By Lemma 6.5 and Corollary 6.5, the minimum possible safe guard time ϕ is given by:

$$\phi = \sigma \cdot \frac{2(s+\mathsf{k}) \cdot \delta^{max}}{1 - 4(s+\mathsf{k}) \cdot \delta^{max}}. \tag{6.7}$$

By summing up the equation (6.6) and (6.7), it follows that:

$$\mathsf{Rec}_{safeSend}(\mathcal{I}[t,t'']) \leq 1 - 4(s+\mathsf{k}) \cdot \delta^{max},$$

which contradicts the equation (6.5). □

6.7 Case Study: A Wireless Fire Alarm System

This section presents a study of the effectiveness of the given formal approach in this chapter, to optimize guard time *analytically*, under real-world conditions. A wireless fire alarm system – which has been developed using a classical-engineering approach – is considered. It is shown that the computed guard times have significantly shorter lengths than the one given in the fire alarm system design. It is also shown that by adding further restrictions, e.g., on the system's topology and the slot assignment, guard time can safely be reduced even further, which in turn increases the recurrence of *safeSend*.

6.7.1 The Wireless Fire Alarm System

The considered system is a Wireless Fire Alarm System – shortly WFAS – in the sense of EN 54-25 [DIN05]. A WFAS is a wireless sensor network consisting of a central unit besides nodes including sensors and repeaters. Sensors are supposed to send information (messages) to the central unit. If the physical distance between a sensor and the central unit is large, repeaters are used to forward messages to the central unit. Figure 6.6 illustrates an example of a WFAS topology. The nodes communicate with each other

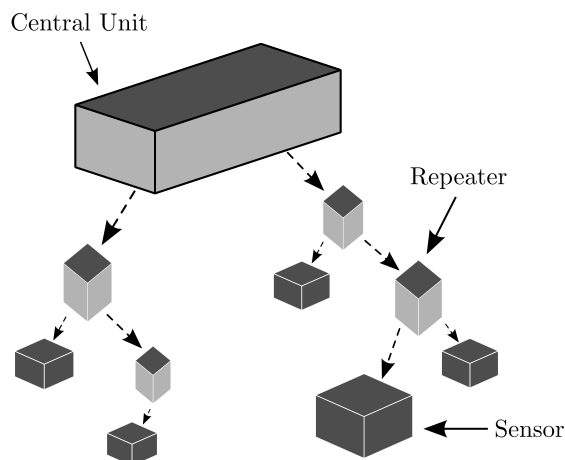


Figure 6.6: Wireless fire alarm system

via radio signals over a shared channel using TDMA. The communication scheme in this network, together with clock synchronization, is same as in the given model in the previous sections. Note that in Figure 6.6, the arrows are directed from parents to children; they do not denote the direction of messages, but the direction of timestamped acknowledgements. Each node is equipped with a hardware clock. It is assumed here that each node other than the central unit is assigned exactly one slot per frame.

There are restrictions on the network size: The maximum number of sensors and repeaters is 126, and the maximum number of repeaters linking a sensor to the central unit is 5. The slot length is 25 ticks and slots are further divided into sub-intervals. This is shown in Figure 6.7.

Sensors and repeaters in the WFAS's are typically battery powered and EN 54-25 in particular requires a minimum battery lifetime [DIN05]. The overall energy consumption

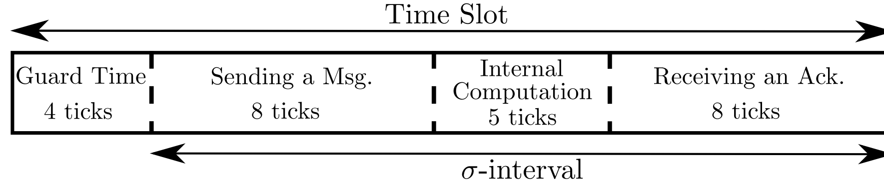


Figure 6.7: Time slot of the WFAS

is dominated by energy consumption during both, sending and listening phases. Therefore, energy efficiency is a prominent issue with EN 54-25 compliant WFAS.

Environmental conditions like temperature and battery voltage lead to a variation in the crystal frequencies of the clocks, and thus *clock drift* may be exhibited at sensors and repeaters. The frequency of the clock modules used in sensor and repeater nodes is 32.768 kHz, the accuracy is 20 ppm[1] under the environment conditions specified by the manufacturer of the clock modules.

WFAS's are safety-critical and have to satisfy response deadlines. Since message collision and loss may lead to a communication failure, they have to be avoided. The clock drift issue is tolerated using the guard time. The purpose of the guard time in the WFAS is only to avoid collision and loss due to clock drift, but not communication failures during a slot caused by, e.g., a weak signal. Communication failures of the latter kind are treated in the WFAS as part of the communication protocol during the slot and are not considered here.

6.7.2 Modelling the WFAS

The WFAS belongs to the class of systems that can be formalized by the model given in Section 6.3. The topology is modelled as a tree $\mathcal{T} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{\mathsf{cu}(\mathcal{T})\} \dot\cup \mathsf{Sn}(\mathcal{T})$, such that $\mathsf{cu}(\mathcal{T})$ (central unit) is the tree root, and $\mathsf{Sn}(\mathcal{T})$ is the set of sensors *and repeaters*. Here, there is no distinguishing between sensors and repeaters, since both may exhibit the same amount of clock drift. Any topology has a maximum depth of 6, since 5 repeaters at most are allowed to be on one path of the topology. The size of $\mathsf{Sn}(\mathcal{T})$ is up to 126. There is no restriction on any subtree size s, therefore, $1 \leq s \leq |\mathsf{Sn}(\mathcal{T})|$.

In WFAS, the domain of the clock values is $\mathbb{N}_0$ because time is discretized into *ticks*. However, the clock of a node v in an evolution $\mathcal{I}$ at time t is modelled by the given function $\mathsf{clk}_n^{\mathcal{I}}(t) : \mathsf{Time}\ (= \mathbb{R}_0^+)$, i.e. by real numbers. By using real numbers, one can compute a precise value of an optimal guard time. If the computed value has fractions, it can be rounded to the next natural number reflecting guard time length in ticks. Since the clock accuracy is 20ppm, the maximum drift rate δ^{max} equals 0.00002.

Since sensors and repeaters are assigned slots bijectively, and clocks are synchronized by the time-stamped acknowledgments with one synchronization point for each node in each assigned slot, the evolutions of the WFAS are scheduled and synchronized.

Given Figure 6.7, the slot length σ, excluding guard time, equals 21 ticks, and the guard time in this system is only one interval added to the beginning of a slot, while the guard time in the formal model differs. This issue is treated in the following section.

[1] ppm abbreviates 'parts per million'; 1 ppm = 10^{-6}.

6.7.3 Guard Time Optimization

In the following parts, the formal approach is applied to derive guard times for several situations of the WFAS. Following Corollary 6.4, the number of nodes (which is equal to the number of slots per frame) and the topology depth are directly proportional with the optimal guard time. Therefore, each considered WFAS has the maximum values of its sizes as a worst case wrt. the number of slots per frame k and the depth d. The value of k is 126, and d equals 6.

Note that in the following parts, each situation is treated separately. However, there may exist cases that combine more than one situation. For example, a requirement might be to compute optimal guard time treating only message collision and for a best case assignment. In this case, the methodologies for both situations can be combined.

Safe Guard Time

Let d be the topology depth. By Corollary 6.4, a safe guard time exists iff $\delta^{max} < \frac{1}{4(d(\mathsf{k}-1)+2)}$. Given that $\delta^{max} = 0.00002$, $\mathsf{k} = 126$, and $d = 6$, then:

$$0.00002 < \frac{1}{4(6 \cdot (126-1)+2)} = 0.000332447,$$

thus, a safe guard time exists. By Corollary 6.4, and given that σ equals 21 ticks, the optimal guard time wrt. δ^{max} is:

$$\phi_{opt} = 21 \cdot \frac{2(6(126-1)+2) \cdot 0.00002}{1 - 4(6(126-1)+2) \cdot 0.00002} \approx 0.67 \tag{6.8}$$

The safe guard time is, then, two intervals added to the beginning and to the end of a slot, respectively, where each interval is 0.67 ticks. By rounding this length to 1 tick, the overall length of both intervals is 2 ticks, implying that the guard time employed by the WFAS (4 ticks, cf. Figure 6.7) is reduced to half of its length.

Guard Time Treating Only Message Collision

The protocol employed by the WFAS does not utilize guard time to treat message loss caused by clock drift, and these losses are treated by the employed protocol together with other communication failures caused by, e.g., other users of the frequency band. Yet the derived guard time in the previous section treats message loss caused by clock drift. Thus, it is shown how guard time can be extended, optimally, to treat only collision.

By Theorem 6.1, treating message loss requires having smaller bounds on clock drift than the bounds allowed by message collision, which implies that the required guard time for treating only message collision is smaller. Given a topology $\mathcal{T}$, by Theorem 6.1, any evolution $\mathcal{I}$ over $\mathcal{T}$ does not have any message collision if

$$\forall\, v \in \mathsf{Sn}(\mathcal{T}), t \in \mathsf{Time} \bullet |\varrho_v^{\mathcal{I}}(t)| \leq \phi. \tag{6.9}$$

By Corollary 6.2, the maximum clock drift for an assignment assign is given by:

$$\varrho_{\omega,\delta^{max}}^{max}(\mathsf{assign}) = (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1) \cdot \omega \cdot \delta^{max}. \tag{6.10}$$

By the equations (6.9) and (6.10):

$$
\begin{aligned}
& \delta^{max} \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1) \cdot (2\phi + \sigma) \cdot \leq \phi && (6.11)\\
\Longleftrightarrow\ & \delta^{max} \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1) \cdot \sigma && (6.12)\\
& \leq (1 - 2 \cdot \delta^{max} \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1)) \cdot \phi. && (6.13)
\end{aligned}
$$

By distinguishing the two cases: $1-2\cdot(\mathcal{D}_{\mathsf{assign}}+\mathsf{k}+1)\cdot\delta^{max} > 0$ and $1-2\cdot(\mathcal{D}_{\mathsf{assign}}+\mathsf{k}+1)\cdot\delta^{max} \leq 0$, it follows that a safe guard time exists only if $1-2\cdot(\mathcal{D}_{\mathsf{assign}}+\mathsf{k}+1)\cdot\delta^{max} > 0$. The optimal safe guard time is given by:

$$
\phi_{opt} = \sigma \cdot \frac{(\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1) \cdot \delta^{max}}{1 - 2 \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1) \cdot \delta^{max}}. \qquad (6.14)
$$

Equation (6.14) computes an optimal guard time for a given assignment assign. For the worst case assignment, by Corollary 6.4 and (6.14), an optimal guard time (which is safe for any assignment) can be computed by the equation:

$$
\phi_{opt} = \sigma \cdot \frac{(d(\mathsf{k}-1)+2) \cdot \delta^{max}}{1 - 2(d(\mathsf{k}-1)+2) \cdot \delta^{max}}. \qquad (6.15)
$$

For $d = 6$, $\mathsf{k} = 126$, $\sigma = 21$, and $\delta^{max} = 0.00002$, it follows by (6.15) that:

$$
\phi_{opt} = 21 \cdot \frac{(6(126-1)+2) \cdot 0.00002}{1 - 2(6(126-1)+2) \cdot 0.00002} \approx 0.33. \qquad (6.16)
$$

The computed guard time is, then, two intervals, where each has a length of ≈ 0.33 tick, which is approximately half of the length of the guard time that is used to treat also message loss, given the same parameters.

Recall from Section 6.7.1 that in the WFAS, the guard time is added as one interval to the beginning of each slot, to treat only message collision, while the computed guard time is two intervals that are added to the beginning and to the end of each slot, where each interval has a length of 0.33 tick. By Figure 6.5, the computed guard time can be adapted to the design of the WFAS; i.e. the two intervals are summed up and added as one interval to the beginning of each slot. The interval length is $0.33 + 0.33 = 0.66$ rounded to 1 tick, and added to the beginning of each slot.

Guard Time for Best Assignments

Lemma 6.5 states the restrictions on the best case assignments. If any evolution adheres to the restrictions initially, or if a self-stabilizing algorithm produces assignments under the restrictions, then the optimal guard time for best assignments can provide collision- and loss-free evolutions.

For best assignments, the size of the maximal subtree s is a parameter in the computation of the guard time. Detailed computations of optimal guard time for best assignments are shown for two maximal subtree sizes: (1) an assignment where $s = \mathsf{k} = 126$; i.e. there is only one subtree, and (2) where $s = \frac{\mathsf{k}}{6} = 21$; i.e. a quite restrictive condition where there are possibly many subtrees, but the size of the maximal subtrees is only about a sixth of the components. The values of s are chosen to show how the optimal guard time can be reduced by restricting the size of the maximal subtree.

By Corollary 6.5, the guard time for the best case is given by:

$$\phi_{opt} = \sigma \cdot \frac{2(s+\mathsf{k}) \cdot \delta^{max}}{1 - 4(s+\mathsf{k}) \cdot \delta^{max}}. \tag{6.17}$$

Case $s = 126$: by Corollary 6.5, given a best assignment, there exists a guard time wrt. δ^{max} iff $\delta^{max} < \frac{1}{4(s+\mathsf{k})}$. Given that $\delta^{max} = 0.00002$, $\mathsf{k} = 126$, and $s = 126$, then:

$$0.00002 < \frac{1}{4(126+126)} = 0.000992063.$$

An optimal safe guard time exists wrt. δ^{max} given the best case assignment where $s = 126$. By the equation (6.17), the optimal safe guard time is:

$$\phi_{opt} = 21 \cdot \frac{2(126+126) \cdot 0.00002}{1 - 4(126+126) \cdot 0.00002} = 0.216035271. \tag{6.18}$$

Case $s = 21$

Similar to the case $s = 126$, for $\delta^{max} = 0.00002$, $\mathsf{k} = 126$, and $s = 21$, an optimal guard time exists iff

$$0.00002 < \frac{1}{4(21+126)} = 0.00170068.$$

An optimal guard time exists wrt. δ^{max} given the best case assignment where $s = 21$. By the equation (6.17):

$$\phi_{opt} = 21 \cdot \frac{2(21+126) \cdot 0.00002}{1 - 4(21+126) \cdot 0.00002} = 0.124949405. \tag{6.19}$$

By the equations (6.18) and (6.19), it is obvious that the case $s = 126$ requires (as expected) a larger optimal guard time than the one in case $s = 21$.

For a deeper view on the best case assignments, the possibility of having a reasonable length of optimal guard time, if the clocks' drift rate is large, is analyzed.

Guard Time Treating Other Sorts of Message Loss

In the former sections, it is assumed that there is no message loss due to radio signal issues because it is assumed that there is one synchronization point within each slot. In practice, messages with timestamps can get lost due to e.g. signal weaknesses and delays in message delivery. In such cases, the computed guard times in the previous sections are in general not safe. If one time-stamped acknowledgment is lost, then the one in the next frame may reach its target node too early or late due to continuous clock drift in the node.

Consider a node v at depth d' which is not synchronized within its assigned slot. The clock of the node may continue to drift for one additional frame until the next synchronization point. This is the same situation as if the node had depth $d' + 1$. More general, if v misses all synchronization points during l adjacent frames, then v may drift as if v has a depth of $d + l$.

By Corollary 6.4, given a tree of depth d, an arbitrary assignment, and an evolution, if any node may miss all synchronization points during at most l adjacent frames, then a safe guard time can be computed as follows:

$$\phi_{opt} = \sigma \cdot \frac{2((d+l)(\mathsf{k}-1)+2) \cdot \delta^{max}}{1-4((d+l)(\mathsf{k}-1)+2) \cdot \delta^{max}}. \tag{6.20}$$

For example, given a topology with $d = 6$, $\mathsf{k} = 126$, $\omega = 21$, $\delta^{max} = 0.00002$, and $l = 5$, then by (6.20):

$$\phi_{opt} = 21 \cdot \frac{2((6+5)(126-1)+2) \cdot 0.00002}{1-4((6+5)(126-1)+2) \cdot 0.00002} \approx 1.3.$$

Interestingly, Equation (6.20) provides an option to compute the number of subsequent message losses that can be tolerated by a given guard time. For example, given a WFAS with the maximum sizes, and the original guard time employed by the system is $2\phi = 4$ ticks (cf. Figure 6.7), i.e. $\phi = 2$, then the number of tolerated subsequent message losses is computed by the equation (6.20) as:

$$2 = 21 \cdot \frac{2((6+l)(126-1)+2) \cdot 0.00002}{1-4((6+l)(126-1)+2) \cdot 0.00002}$$
$$l \approx 10.$$

This implies that 9 subsequent message losses can be tolerated by the guard time given by the system design.

6.7.4 Energy Consumption

This part shows a comparison of systems employing the computed guard times in Section 6.7.3 wrt. the required time and energy. The time and energy consumption are investigated per frame, since a frame is a periodic cycle of a static length and schedule. In the WFAS, the energy consumptions by any node during sleep, sending, and listening modes are 60μA, 65mA, and 40mA, respectively.

By fundamental theorems of physics, an *Ampere* equals ${}^{Coulombs}/_{Second}$. Since slots and guard time are expressed as ticks, and given the clock speed as ticks per second, energy is expressed as *Coulombs* per *Tick* (CpT). The clock speed is 32.768kHz. The energy consumptions by any node during sleep, sending, and listening modes are, then, $2 \cdot 10^{-9}$CpT, $\approx 2 \cdot 10^{-6}$CpT, and $\approx 1.2 \cdot 10^{-6}$CpT, respectively.

The same system parameters as in Section 6.7.3 are used: the number of slots per frame k is 126, and an σ-interval (slot excluding guard time) is 21 ticks. Given that σ-intervals have constant length, the variable in the comparison is only the guard time length. Therefore, it is assumed that the required energy during σ-intervals of all slots in one frame is e.

Recall that the number of nodes (including central unit) is 127. During the guard time of each slot, there exist two nodes in listening mode, where the other 125 are in sleep mode. Recall that the guard time is two intervals, each of length ϕ. The required extra energy for guard time during one frame is:

$$2\phi \cdot ((125 \cdot 2 \cdot 10^{-9}) + (2 \cdot 1.2 \cdot 10^{-6})) = 2.65 \cdot 10^{-6} \cdot 2\phi \text{ CpT}. \tag{6.21}$$

Table 6.1 presents the required time and required energy by all nodes in one frame, while employing the guard times given in Sections 6.7.1 and 6.7.3.

	Guard Time 2ϕ (ticks)	Frame Length (ticks)	Energy Consumption (Coulombs)
Original	4.00	3150.00	$10.6000 \cdot 10^{-6} + e$
Optimal for any assignment	1.34	2814.84	$3.5510 \cdot 10^{-6} + e$
Opt. for any assignment, only collision	0.66	2729.16	$1.7490 \cdot 10^{-6} + e$
Opt. for best assignments, $s = 126$	0.44	2701.44	$1.1660 \cdot 10^{-6} + e$
Opt. for best assignments, $s = 21$	0.25	2677.50	$0.6625 \cdot 10^{-6} + e$

Table 6.1: Optimal safe guard times – both left and right intervals – and their effect on time and energy consumption.

6.7.5 Cross-checking Derived Guard Times

As shown in the previous sections, the model can be extended to match various cases and assumptions. This, however, requires to confirm that the extended model or equation is correct, especially if the extension involves many parameters. This section provides a method to cross-check the correctness of the results obtained by the formal approach using automatic verification tools.

A model of parents and children with listen and send phases is provided. Correctness checking then amounts to checking whether two children with adjacent slots can ever assume their send phase at the same time point (collision) and whether there is a point in time where a child is in the send phase while its parent is not listening (loss).

Note that a faithful model of clock drift involves hybrid aspects because clocks evolve at different speeds and the clock speed may even change over time. This behavior is over-approximated using timed automata [AD94], in which clocks evolve at uniform speeds.

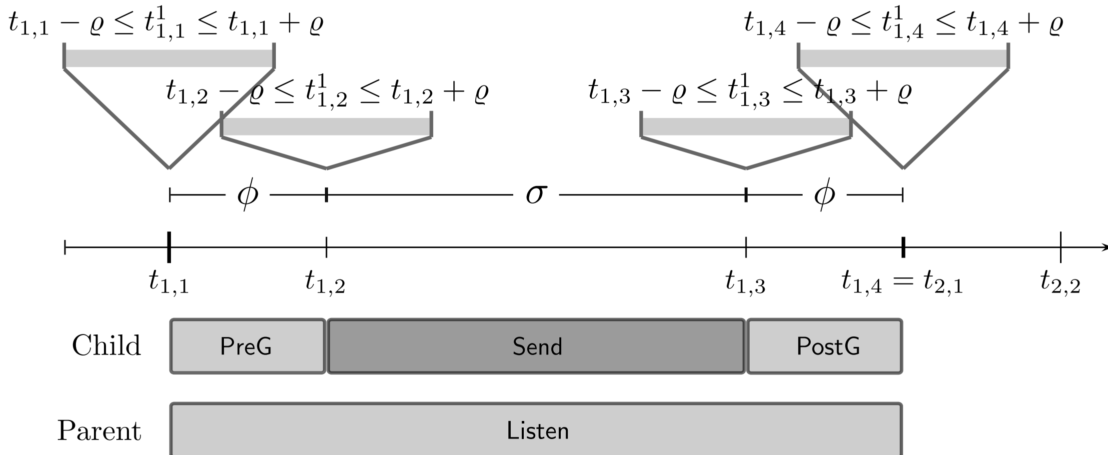
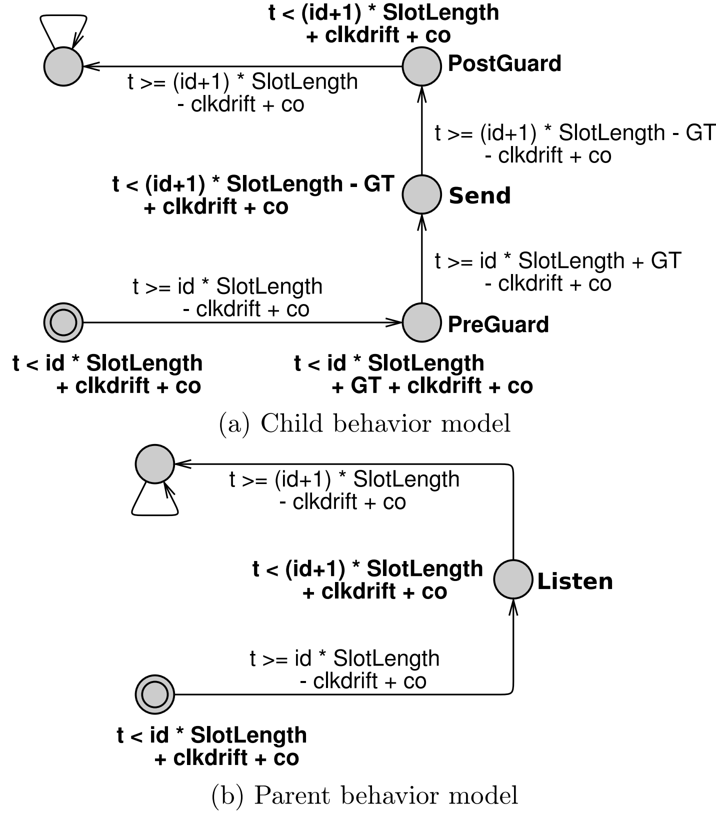


Figure 6.8: Verification of derived guard times



(a) Child behavior model

(b) Parent behavior model

Figure 6.9: Uppaal models – timed automata

The idea underlying the model is illustrated in Figure 6.8. It shows one slot from $t_{1,1}$ to $t_{1,4}$ as seen by the central unit which provides the reference time. If the clock of a sensor runs in perfect synchronization with the reference time, it should be in its left guard interval from $t_{1,1}$ to $t_{1,2}$ (PreG), in its sending phase from $t_{1,2}$ to $t_{1,3}$ (Send), and in its right guard interval from $t_{1,3}$ to $t_{1,4}$ (PostG). If the clock of this sensor's parent runs in perfect synchronization with the reference time, then the parent should be listening from $t_{1,1}$ to $t_{1,4}$ (Listen).

If the sensor's clock drifts with a maximum difference ϱ to the reference time, then it enters phase PreG at $t_{1,1} - \varrho$ the earliest and $t_{1,1} + \varrho$ the latest.

This is indicated by the triangle-like areas in Figure 6.8. The time where the sensor enters PreG is denoted by $t^1_{1,1}$ in Figure 6.8. The same observation holds for leaving PreG, and for entering and leaving Send and PostG, respectively, and for the parent for phase Listen.

This behavior with clock drift is modelled by the timed automata templates shown in Figure 6.9. The initial location of the sensor – cf. Figure 6.9a – models the sensor being idle during a frame until its slot is reached at time $id \cdot \omega$ where $id \in \mathbb{N}_0$ gives the number of its slot. This time corresponds to $t_{1,1}$ above. The slot length ω is called SlotLength in Figure 6.9. The sensor moves to its PreGuard phase at $id \cdot \omega - \varrho$ (inclusive) the earliest and at $id \cdot \omega + \varrho$ (exclusive) the latest. Here, the clock x of the sensor automaton provides the reference time. Clock drift is modelled by relaxing the guards

by ϱ in both directions. Similarly, the sensor moves to its Send phase at $id \cdot \omega + \phi - \varrho$ (inclusive) the earliest and at $(id + 1) \cdot \omega + \phi + \varrho$ (inclusive) the latest where ϕ and ϱ are called GT (abbrev. guard time) and clkdrift in the model, respectively. The behavior of the parent is modelled similarly – cf. Figure 6.9b.

As clocks of timed automata start with value 0, in order to include the behavior of fast clocks, all times are shifted by adding a constant co to model evolutions where the sensor enters PreGuard *before* the beginning of the slot as indicated by the reference time. The constant co must be bigger or equal to ϱ.

Note that the model in Figure 6.9 does not allow evolutions where triangle-like areas in Figure 6.8 overlap, i.e. where for example $t^1_{1,2} < t^1_{1,1}$. This is valid because in the real system, the three phases are entered one after the other by a sequential implementation. The model is an over-approximation of the timed evolutions over a topology if a least upper bound δ^{max} on the magnitude of the drift rates in the evolution is considered. For example, the model allows arbitrarily short dwelling times for location PreGuard for certain values of ϕ and ϱ. This would correspond to a clock running slow until entering PreGuard and drifting with an arbitrarily large speed in or to leave PreGuard early. Yet if the over-approximation is safe, it is concluded that the chosen guard time is safe.

By symmetry, it is sufficient to check a model instance consisting of two children and their parents. It is considered that there is a child/parent pair for slot id = 0 and one for adjacent slot id = 1 with slot length SlotLength = 50, maximal clock drift clkdrift = 2, and shifting constant co = 2.

To check that this model instance adheres to the properties implied by Theorem 6.1, the Uppaal model checker [BDL04] is used, which is the typical model checker for timed automata. The properties to be checked are as follows:

1. If GT equals clkdrift (here: GT = 2) then no evolution exhibits message collision. This property is checked by the following query:

   ```
   A[] !(Child(0).Send && Child(1).Send)
   ```

   The query requires that for all evolutions, both sensors are not at location Send at the same time. The query is satisfied.

2. If GT equals $2 \cdot$ clkdrift (here: GT = 4) then no evolution exhibits message loss. This property is checked by the following query:

   ```
   A[](Child(0).Send imply Parent(0).Listen)
    && (Child(1).Send imply Parent(1).Listen)
   ```

   The query requires that for all evolutions, if a child is in location Send, then its parent is in location Listen. The query is satisfied.

3. If GT equals $2 \cdot$ clkdrift, then GT is optimal. This can be checked by extending the time period in which the transition to location Send can be taken. Technically, '$<$' is replaced in the invariant of the initial location with '$\leq$'. Then the above queries are checked. The queries are not satisfied in this case, implying that the guard time 4 is optimal in this example.

6.8 Remarks

This section highlights some remarks of the presented work in this chapter.

The formalism of the wireless sensor network in the sense of EN-54-25 [DIN05], including the restrictions, provides the ability to do rigorous and thorough analysis of properties over networks of such. The main question, that is considered is, how does the order of TDMA slot assignment impact the clock synchronization precision. This question is analyzed in the scope of the model, and the main results of the analysis indicate the worst and best slot assignment orders wrt. the clock synchronization precision. Some additional mathematical analysis provided an equation system for optimizing guard time for each slot assignment. Additionally, a case study of a wireless fire alarm system is considered. For this case, the formal model is easily extended to match real-world cases. After that, a method for cross checking the correctness of the approaches – based on automatic verification tools – is provided.

The following two remarks concern the results:

- In regard to the communication scenario between child and parent within one slot, the assumption in the model is abstract: a synchronization point may be at any position in the slot. In practice, such a point would exist usually at the end of the slot, after receiving the acknowledgement. If this could be somehow guaranteed, guard time could be further optimized.

- The worst assignment, which is shown to be the worst according to the clock synchronization precision, might still be efficient wrt. the direction of message delivery. The reason is that in this assignment, the sensors of the longest path are assigned reverse slots, and therefore, a message can be delivered from the bottom of the tree up to the central unit within one frame. From this point, if some design requires that one path is assigned reverse slots, it is already a worst assignment wrt. clock synchronization precision, and it would be rather good if all other paths are assigned reverse slots to guarantee fast bottom-up message delivery.

Finally, the next question that may be considered is, whether self-stabilization wrt. a best assignment for this network is achievable. In general, for such networks, if e.g. a communication disturbance happens, the sensors can stop sending messages, and go into a listening mode until they receive some acknowledgement. Naturally, due to the tree structure, the acknowledgement may start from the central unit and goes down to the leaves. The next question would be, whether the PIF algorithms designed in Chapter 5 may provide hints about the design of a self-stabilizing system for this issue. The communication nature between a process and its neighbors in the PIF algorithms is similar to the presented model in this chapter.

7 Automatic Verification of Recurrence Properties

This chapter investigates the possibility of verifying recurrence properties using automatic verification tools. The basic challenge in this case is that recurrence properties are defined as ratios of conditions over infinite executions, which hardens automatic verification. The aim in this chapter is to simplify and enable automatic verification of recurrence properties by basically overcoming the issue of infinite executions. This is achieved by exploiting the fact that each configuration in a self-stabilizing system is considered to be initial, which implies that checking all possible execution prefixes having some fixed length is sufficient to know whether the system violates the property. Some of this chapter's content has appeared in [6] of the author's publications.

The chapter is structured as follows. Section 7.1 provides an overview of using automatic verification for verifying self-stabilizing systems. Section 7.2 presents the problem statement and assumptions. Next, Section 7.3 presents an approach that enables and simplifies the use of automatic verification tools for verifying recurrence properties. Section 7.4 explains the usefulness of model checking for this case. Finally, Section 7.5 presents a case study.

7.1 Automatic Verification of Self-Stabilizing Systems

The design and verification of self-stabilizing distributed systems is known to be tough. There are two major reasons for it. First, each configuration is considered to be an initial configuration. The problem gets more complicated if the system has an infinite configuration space. Second, executions can be infinite. This makes it hard to verify e.g. liveness properties and even convergence.

Model checking [CGP99, BK08] is a useful technique for systems having a finite configuration space, and classical self-stabilization can be verified by simple model checking, given enough time and memory. However, for systems having an infinite configuration space, simple model checking may not succeed, and additionally, abstraction techniques are needed. In many cases, the initial configurations are classified into finite classes, and the properties to be checked might require to be reformed to match the classes. In other cases, other techniques are used, like theorem provers and symbolic model checkers. If the abstraction is robust enough to restrict the infinite space and executions, classical self-stabilization can still be checked.

Verifying self-stabilization wrt. recurrence properties – con_Δ- convergence and warmup – via model checking is not straightforward. This is due to two facts:

- Recurrence properties are defined over infinite executions. A minimum ratio Δ of configurations satisfying con should always exist in each execution prefix, after the

system stabilizes. A model checker may not be able to detect when to stop checking this property in an infinite execution. This holds for both $\mathsf{con}_\triangle$-convergence and $\mathsf{con}_\triangle$-warmup.

- If a $\mathsf{con}_\triangle$-convergence time of c steps is *guaranteed* by an execution, in contrast to the classical self-stabilization, the initial configuration of the execution suffix that satisfies $\mathsf{con}_\triangle$ within the first $\mathsf{c}+1$ configurations is usually unknown.

The aim of this chapter is to find an abstraction technique that simplifies verifying recurrence properties by overcoming the issue of infinite executions. The following part summarizes related work.

7.1.1 Related Work

Using formal and automatic verification tools for verifying self-stabilizing systems is growing recently. Automatic verification is mostly used to support the design and evaluation of self-stabilizing systems.

Early work in this scope is given by Theel in [The00a, The00b, The01]. It aims to automate verification using the concept of Lyapunov Functions [Lya07], which is intensively used in the area of control theory: Lyapunov functions are a sort of ranking functions that are applied to continuous and hybrid systems. A following work [ODT05] provides a convergence verification approach that adopts certain techniques for verification of particular hybrid systems to verify self-stabilization in distributed systems. The challenge in these approaches is to find a suitable model for distributed systems and the verification techniques used for hybrid systems. Other transformational approaches, e.g. [KT10], adopt the use of ranking functions for systems running under the central scheduler to the distributed scheduler.

Other work focuses on analyzing performance aspects of self-stabilizing systems. Examples are [NKM06, DTW06, FBT13]. The work of [NKM06] considers the occurrence of transient faults during the convergence, and their effect on the convergence time. The approach of [DTW06] defines and applies fault tolerance measurements, such as availability, to evaluate self-stabilizing systems, also under the assumption of ongoing transient faults. In [FBT13], the authors use a metric for measuring the expected mean value of the system's convergence time. This value denotes the average case of the convergence time, and is computed by probabilistic model checking. These approaches and others are basically aimed to evaluate the performance of self-stabilizing systems using formal methods and automatic verification, which is found to be useful.

Some recent work considers using automatic verification to support designing self-stabilizing systems. Examples are [KE14, FB14, KKM15]. In [KE14], the authors present a formal method for algorithmic design of self-stabilizing systems based on variable superposition and backtracking search. In [FB14], the authors exploit SMT solvers [dMB11] for synthesizing self-stabilizing algorithms. In [KKM15], the authors use distributed local verification and present an algorithmic proof labelling schemes to support the design of self-stabilizing algorithms. The usefulness of [KKM15] for optimizing time and space complexity of a self-stabilizing minimum spanning tree (MST) is shown.

In contrast to related work, this work exploits model checking for proving the absence of counterexamples wrt. recurrence properties, to conclude that the analyzed system satisfies the recurrence property.

7.2 Problem Statement and Assumptions

Recall that the basic aim in this chapter is to overcome the challenge of having infinite executions, while applying automatic verification. The challenge arises from the fact that the property con_Δ is defined over infinite executions, and the recurrence of con may differ among subexecutions of an execution. If a system does not satisfy con_Δ-convergence or con_Δ-warmup in a given number of steps, then there exists a finite subexecution of an execution that does not satisfy the property; i.e. a counterexample.

7.2.1 Problem Statement

The issue considered in this work is, whether there exists always a counterexample of some fixed length for any system that does not satisfy the corresponding property.

Problem 7.1. Given a system Ω that does not satisfy con_Δ-convergence in c steps (or con_Δ-warmup in w steps), identify the minimum length of a counterexample that is an execution prefix of an execution in Ω.

The contribution of this chapter is as follows:

- It is shown that for each system that does not satisfy con_Δ-convergence in c steps (resp. con_Δ-warmup in w steps), there exists an execution prefix having a length of $\mathsf{c}+1$ (resp. in $[\mathsf{w}+1, 2\mathsf{w}+1]$) configurations that is a counterexample.
- For systems having a finite configuration space (but not necessarily finite executions), it is shown how model checking is used to verify the properties.
- As a case study, the model checker nuXmv [CCD+14] is applied to analyze the recurrence of granting a privilege (i.e. the service time) of a mutual exclusion algorithm, namely Algorithm 4.2 from Chapter 4, executed over many topologies.

7.2.2 Assumptions

For the model checking parts (Section 7.4 and 7.5), for simplicity, it is assumed that the systems have finite configuration spaces. For systems with infinite configuration spaces, there are abstraction techniques applied by some model checkers, which are out of the scope of this work.

7.3 Counterexamples of Fixed Length

This section provides an approach for simplifying the procedure of proving recurrence properties, by checking the absence of counterexamples of fixed length. The following definition specifies the meaning of counterexample for con_Δ-convergence and con_Δ-warmup. To distinguish both cases, “counterexample” is written to refer to the

convergence property, and "wu-counterexample" is written to refer to the warmup property.

Definition 7.1 (Counterexample)**.** Let con be a condition, $\Delta \in [0,1] \subset \mathbb{R}$, and $\mathsf{c}, \mathsf{w} \in \mathbb{N}_0$.

- A **counterexample wrt.** $(\mathsf{con}, \Delta, \mathsf{c})$ is a finite execution (prefix) Ξ that does not satisfy con_Δ-convergence in c steps.[1]
- A wu**-counterexample wrt.** $(\mathsf{con}, \Delta, \mathsf{w})$ is a finite execution (prefix) Ξ that does not satisfy con_Δ-warmup in w steps. ♢

The following definition introduces the notion of *minimal* counterexample, which denotes a counterexample that has no strict subexecution that is a counterexample.

Definition 7.2 (Minimal Counterexample)**.** Let con be a condition, $\Delta \in [0,1] \subset \mathbb{R}$, and $\mathsf{c}, \mathsf{w} \in \mathbb{N}_0$.

- A counterexample Ξ wrt. $(\mathsf{con}, \Delta, \mathsf{c})$ is said to be *minimal* iff there exists no strict subexecution Ξ' of Ξ such that Ξ' is a counterexample.
- A wu-counterexample Ξ wrt. $(\mathsf{con}, \Delta, \mathsf{w})$ is said to be *minimal* iff there exists no strict subexecution Ξ' of Ξ such that Ξ' is a wu-counterexample. ♢

The interesting point that enables finding a counterexample, is that each configuration in a self-stabilizing system is an initial configuration of some execution, and having a minimal counterexample Ξ' as a subexecution of any execution implies that Ξ' is indeed a separate execution (prefix) in the system. Consequently, systems having a finite configuration space have a finite number of minimal counterexamples.

In the remainder of this section, it is proven that for each system that does not satisfy con_Δ-convergence in c steps (resp. con_Δ-warmup in w steps), there exists a minimal counterexample (resp. wu-counterexample) Ξ', whose length is $\mathsf{c}+1$ (resp. in $[w+1, 2w+1]$).

From now on, for any finite execution Ξ, the number of configurations satisfying a condition con in Ξ is denoted by $sat_{\mathsf{con}}(\Xi)$. This, by Definition 3.1 (recurrence definition), entails that the recurrence of con in Ξ ($\mathsf{Rec}_{\mathsf{con}}(\Xi)$) is equal to $\frac{sat_{\mathsf{con}}(\Xi)}{\mathsf{length}(\Xi)}$.

The following is a basic lemma, that regards concatenating two subsequent executions, satisfying particular recurrence properties.

Lemma 7.1. Let $\Xi : \gamma_i, \ldots, \gamma_{j-1}, \gamma_j, \ldots, \gamma_{u-1}$ be a finite execution, such that $\mathsf{Rec}_{\mathsf{con}}(\gamma_i, \ldots, \gamma_{j-1}) \geq \Delta$, and $\forall\, j \leq s \leq u-1 \bullet \mathsf{Rec}_{\mathsf{con}}(\gamma_j, \ldots, \gamma_s) \geq \Delta$. The following statement holds:

$$\forall\, j \leq s \leq u-1 \bullet \mathsf{Rec}_{\mathsf{con}}(\gamma_i, \ldots, \gamma_s) \geq \Delta.$$

[1] "counterexample" is written without "wrt. $(\mathsf{con}, \Delta, \mathsf{c})$" if it is clear from the context.

Proof. The cases where $\mathsf{length}(\gamma_i, \ldots, \gamma_{j-1}) = 0$ or $\mathsf{length}(\gamma_j, \ldots, \gamma_{u-1}) = 0$ hold trivially. Therefore, it is assumed that $\mathsf{length}(\gamma_i, \ldots, \gamma_{j-1}) \geq 1$ and $\mathsf{length}(\gamma_j, \ldots, \gamma_{u-1}) \geq 1$. By the premises, it follows that

$$\begin{aligned} & \mathsf{Rec}_{\mathsf{con}}(\gamma_i, \ldots, \gamma_{j-1}) \geq \Delta & (7.1)\\ \iff & \frac{sat_{\mathsf{con}}(\gamma_i, \ldots, \gamma_{j-1})}{(j-1) - i + 1} \geq \Delta & \\ \iff & sat_{\mathsf{con}}(\gamma_i, \ldots, \gamma_{j-1}) \geq \Delta(j - i) & (7.2) \end{aligned}$$

Analogous to the derivation of Formula (7.2):

$$\forall\, j \leq s \leq u - 1 \bullet sat_{\mathsf{con}}(\gamma_j, \ldots, \gamma_s) \geq \Delta(s - j). \tag{7.3}$$

By Definition 3.1 and by Formula (7.2) and Formula (7.3), the following derivation applies:

$$\begin{aligned} \forall\, j \leq s \leq u - 1 \bullet \mathsf{Rec}_{\mathsf{con}}(\gamma_i, \ldots, \gamma_s) &= \frac{sat_{\mathsf{con}}(\gamma_i, \ldots, \gamma_s)}{s - i + 1} \\ &= \frac{sat_{\mathsf{con}}(\gamma_i, \ldots, \gamma_{j-1}) + sat_{\mathsf{con}}(\gamma_j, \ldots, \gamma_s)}{s - i + 1} \\ &\geq \frac{\Delta(j - i) + \Delta(s - j + 1)}{s - i + 1} \\ &\geq \Delta. \end{aligned}$$

□

The following corollary and the lemma show that the length of any minimal counterexample wrt. $(\mathsf{con}, \Delta, \mathsf{c})$ is $\mathsf{c} + 1$.

Corollary 7.1. For each counterexample Ξ wrt. $(\mathsf{con}, \Delta, \mathsf{c})$, $\mathsf{length}(\Xi) \geq \mathsf{c} + 1$.

Proof. It follows by the definitions 3.3 and 7.1. □

Lemma 7.2. For each minimal counterexample Ξ wrt. $(\mathsf{con}, \Delta, \mathsf{c})$: $\mathsf{length}(\Xi) = \mathsf{c} + 1$.

Proof. Corollary 7.1 implies that the length of any counterexample is greater or equal to $\mathsf{c} + 1$. It remains to show that the length of each minimal counterexample is not greater than $\mathsf{c}+1$. By contradiction: assume that there exists a minimal counterexample $\Xi : \gamma_0, \ldots, \gamma_{u-1}$ wrt. $(\mathsf{con}, \Delta, \mathsf{c})$, where $u = \mathsf{length}(\Xi) > \mathsf{c}+1$. By Definitions 3.3 and 7.1, it follows that:

$$\forall\, i \leq \mathsf{c} \bullet \exists\, j \geq \mathsf{c} \bullet \mathsf{Rec}_{\mathsf{con}}(\gamma_i, \ldots, \gamma_j) < \Delta. \tag{7.4}$$

Since Ξ is minimal, by Definition 7.1, each strict subexecution of Ξ is not a counterexample. Let Ξ' be the strict subexecution $\gamma_1, \ldots, \gamma_{u-1}$ of Ξ. Since $\mathsf{length}(\Xi) > \mathsf{c} + 1$, it follows that $\mathsf{length}(\Xi') \geq \mathsf{c} + 1$. By Definition 3.3, this implies that:

$$\exists\, i \leq \mathsf{c} + 1 \bullet \forall\, j \geq \mathsf{c} + 1 \bullet \mathsf{Rec}_{\mathsf{con}}(\gamma_i, \ldots, \gamma_j) \geq \Delta. \tag{7.5}$$

By considering Formula (7.4), the formula holds for any i among $\{1, \ldots, \mathsf{c}\}$. This implies that Formula (7.5) does not hold for any i among $\{1, \ldots, \mathsf{c}\}$. This implies that i may only be $(\mathsf{c} + 1)$ in Formula (7.5); i.e.:

$$\forall\, j \geq \mathsf{c} + 1 \bullet \mathsf{Rec}_{\mathsf{con}}(\gamma_{\mathsf{c}+1}, \ldots, \gamma_j) \geq \Delta. \tag{7.6}$$

By Formula (7.6) and Definition 3.1:

$$\gamma_{\mathsf{c}+1} \models \mathsf{con} \tag{7.7}$$

By Formula (7.4), it follows that $\exists\, j \geq \mathsf{c} \bullet \mathsf{Rec}_{\mathsf{con}}(\gamma_{\mathsf{c}}, \ldots, \gamma_j) < \Delta$. This, by Formula (7.6), Formula (7.7), and Lemma 7.1, implies that:

$$\gamma_{\mathsf{c}} \not\models \mathsf{con} \tag{7.8}$$

A case distinction based on the value of c is done:

1. $\mathsf{c} = 0$. By Definition 7.1 and Formula (7.8), γ_{c} is a counterexample of length $1 = \mathsf{c} + 1$. This contradicts the assumption that Ξ, with $\mathsf{length}(\Xi) > 1$, is a minimal counterexample.

2. $\mathsf{c} > 0$. By assumption, it holds that $\mathsf{length}(\gamma_0, \ldots, \gamma_{\mathsf{c}}) \geq \mathsf{c} + 1$. By minimality of Ξ, the strict subexecution $\gamma_0, \ldots, \gamma_{\mathsf{c}}$ is not a counterexample; i.e. $\exists\, i \leq \mathsf{c} \bullet \mathsf{Rec}_{\mathsf{con}}(\gamma_i, \ldots, \gamma_{\mathsf{c}}) \geq \Delta$. However, since $\gamma_{\mathsf{c}} \not\models \mathsf{con}$, it follows that $i \neq \mathsf{c}$, which implies that:
$$\exists\, i < \mathsf{c} \bullet \mathsf{Rec}_{\mathsf{con}}(\gamma_i, \ldots, \gamma_{\mathsf{c}}) \geq \Delta. \tag{7.9}$$
By Lemma 7.1, Formula (7.6), and Formula (7.9):
$$\exists\, i < \mathsf{c} \bullet \forall\, j \geq \mathsf{c} \bullet \mathsf{Rec}_{\mathsf{con}}(\gamma_i, \ldots, \gamma_j) \geq \Delta. \tag{7.10}$$
There is a contradiction between Formula (7.4) and Formula (7.10). □

Theorem 7.1. If a system Ω does not satisfy con_Δ-convergence in c steps, then there exists a minimal counterexample wrt. $(\mathsf{con}, \Delta, \mathsf{c})$ of length $\mathsf{c} + 1$, that is a prefix of an execution in Ω.

Proof. Since Ω does not satisfy con_Δ-convergence in c steps, then by Definition 7.1, there exists a counterexample Ξ wrt. $(\mathsf{con}, \Delta, \mathsf{c})$ in Ω. If Ξ is minimal, then the theorem holds. Otherwise, by Lemma 7.2, there exists a strict subexecution Ξ' of Ξ, such that $\mathsf{length}(\Xi') = \mathsf{c} + 1$, and Ξ' is a minimal counterexample. By definition of a system, any subexecution of any execution in Ω is indeed an execution prefix of an execution in Ω. The theorem holds. □

To analyze the con_Δ-warmup property, a similar approach is followed. Note that the length of any minimal wu-counterexample lies between $\mathsf{w}+1$ and $2\mathsf{w}+1$. In this case, to find a counterexample via model checking, it is sufficient to check all execution prefixes having any of those lengths.

Theorem 7.2. If a system Ω does not satisfy con_Δ-warmup time of w steps, then there exists a minimal wu-counterexample $\Xi = \gamma_0, \ldots, \gamma_{u-1}$ wrt. $(\mathsf{con}, \Delta, \mathsf{w})$ such that $\mathsf{w} + 1 \leq u \leq 2\mathsf{w} + 1$, and Ξ is an execution (prefix) in Ω.

Proof. By Definition 7.1, and analogous to Corollary 7.1, it follows that any wu-counterexample has a length greater or equal to $\mathsf{w} + 1$. Thus, it remains to show that the length of each minimal wu-counterexample is not greater than $2\mathsf{w} + 1$.

By contradiction: Let $\Xi = \gamma_0, \ldots, \gamma_{u-1}$ be a minimal wu-counterexample, such that $\mathsf{length}(\Xi) = u > 2\mathsf{w} + 1$. By Definitions 3.8, 7.1, and by construction, it follows that $\mathsf{Rec}_{\mathsf{con}}(\Xi) < \Delta$, or equivalently

$$sat_{\mathsf{con}}(\Xi) < u \cdot \Delta. \tag{7.11}$$

Split Ξ into two parts: $\Xi = \xi\beta$ where

$$\Xi = \underbrace{\gamma_0 \cdots \gamma_{\mathsf{w}}}_{\xi} \underbrace{\gamma_{\mathsf{w}+1} \cdots \gamma_{2\mathsf{w}} \cdots \gamma_{u-1}}_{\beta}.$$

Since $\mathsf{length}(\xi) = \mathsf{w} + 1$, by minimality of Ξ, it follows that $\mathsf{Rec}_{\mathsf{con}}(\xi) \geq \Delta$, or equivalently:

$$sat_{\mathsf{con}}(\xi) \geq (\mathsf{w} + 1) \cdot \Delta. \tag{7.12}$$

Likewise, since $\mathsf{length}(\beta) = u - \mathsf{w} - 1 > (2\mathsf{w} + 1) - (\mathsf{w} + 1) = \mathsf{w}$, by minimality of the length of Ξ, it follows that: $\mathsf{Rec}_{\mathsf{con}}(\beta) \geq \Delta$, or equivalently

$$sat_{\mathsf{con}}(\beta) \geq (u - \mathsf{w} - 1) \cdot \Delta. \tag{7.13}$$

Considering that $sat_{\mathsf{con}}(\xi) + sat_{\mathsf{con}}(\beta) = sat_{\mathsf{con}}(\Xi)$, by Formula (7.12) and Formula (7.13), it follows that

$$sat_{\mathsf{con}}(\Xi) \geq (\mathsf{w} + 1) \cdot \Delta + (u - \mathsf{w} - 1) \cdot \Delta, \tag{7.14}$$

which gives

$$sat_{\mathsf{con}}(\Xi) \geq u \cdot \Delta. \tag{7.15}$$

There is a contradiction between Formula (7.11) and Formula (7.15). □

A following example is given to show that $2\mathsf{w} + 1$ is indeed the least upper bound. The following execution Ξ has length $2\mathsf{w} + 1$, and is a minimal wu-counterexample wrt. $(\mathsf{con}, \Delta = \frac{1}{2}, \mathsf{w} = 3)$. Again, $\underline{\gamma}$ indicates that γ satisfies con:

$$\Xi : \gamma_0, \gamma_1, \underline{\gamma_2}, \underline{\gamma_3}, \underline{\gamma_4}, \gamma_5, \gamma_6.$$

7.4 Model Checking Recurrence Properties

This section demonstrates how model checking can be applied to check whether a distributed algorithm fulfills the following properties:

(P1) a con_Δ-convergence time of c steps,

(P2) a con_Δ-warmup time of w steps.

Automatic verification using model checking verifies whether some system model satisfies a property. This amounts to checking whether any reachable configuration from an initial configuration satisfies a condition defined over the configuration. Recall from Section 7.2 the assumption that only systems having a finite configuration space are considered in the following parts.

Since recurrence properties are defined over executions, and model checking verifies conditions over configurations, the basic idea for applying model checking is to add a so-called *observer* process, that runs in parallel to the system, and stores the truth values of con for all configurations in the execution. Next, the model checker verifies if the observer's state satisfies some property, to know whether the recurrence property holds or not.

The model checker NUXMV [CCD+14] is applied in this work. NUXMV is a symbolic model checker for finite and infinite configuration space systems. It provides a flexible input language for modelling distributed algorithms, defined by guarded commands. It applies several verification techniques like bounded model checking (BMC), invariant checking based on binary decision diagrams (BDD), k-induction, interpolation, abstraction refinement, and others. NUXMV handles systems with infinite configuration space by encoding the model checking problem over an abstract configuration space into an SMT problem, that is solved by an SMT solver. NUXMV is an extended version of the model checker NuSMV [CCGR99].

7.4.1 Checking con_Δ-Convergence Time

To check the property (P1), two supplementary objects are added:

- An observer with $\mathsf{c}+1$ registers $b_0, \ldots, b_\mathsf{c} \in \mathbb{B}$.
- A step counter "step," which is initialized with 0 and ranging from 0 to $\mathsf{c}+1$. step is incremented in each step.

The observer stores the first $\mathsf{c}+1$ configurations of an execution Ξ as follows: in each i-th step, the observer assigns the register b_{i-1} the truth value of con in configuration γ_{i-1}. Thus, in the configuration γ_{i+1}, the value of b_i is assigned correctly and reflects whether $\gamma_i \models \mathsf{con}$ is true. Next, the model checker verifies the following formula:

$$\mathtt{step} = \mathsf{c}+1 \longrightarrow \exists\, i \leq \mathsf{c} \bullet \mathit{count}(b_i, \ldots, b_\mathsf{c}) \geq \Delta \cdot (\mathsf{c} - i + 1),$$

where $\mathit{count}(b_i, \ldots, b_\mathsf{c})$ denotes the count of $b_i, \ldots, b_\mathsf{c}$. If the model checker finds an execution leading to a configuration that violates the former formula, then this execution corresponds to an execution Ξ of the system which is a counterexample. Note that since the register b_{i-1} is updated during the i-th step and thus keeps the correct value earliest in configuration γ_i, the formula has to be checked one step later than expected, i.e., when $\mathtt{step} = \mathsf{c}+1$ and not when $\mathtt{step} = \mathsf{c}$.[1]

[1] Although it is possible to do check the formula in the configuration where $\mathtt{step} = \mathsf{c}$, this would result in fair readability.

7.4.2 Checking con_Δ-WarmUp Time

To check (P2), the observer is not needed, and instead, an additional counter "good" is added. Both counters step and good are initialized with 0 and range from 0 to $2\mathsf{w}+1$. The counter good is incremented only in each step where con holds. This enables checking if the following formula holds:

$$\mathsf{w} + 1 \leq \mathtt{step} \wedge \mathtt{step} \leq 2\mathsf{w} + 1 \longrightarrow \mathtt{good} \geq \Delta \cdot \mathtt{step},$$

similar to the previous case.

7.5 Case Study: A Mutual Exclusion Algorithm

This section presents a case study: a study of applying the verification approach – presented above – to a mutual exclusion algorithm, namely Algorithm 4.2 of Chapter 4, which is based on the former work of [BPV08, DG13].

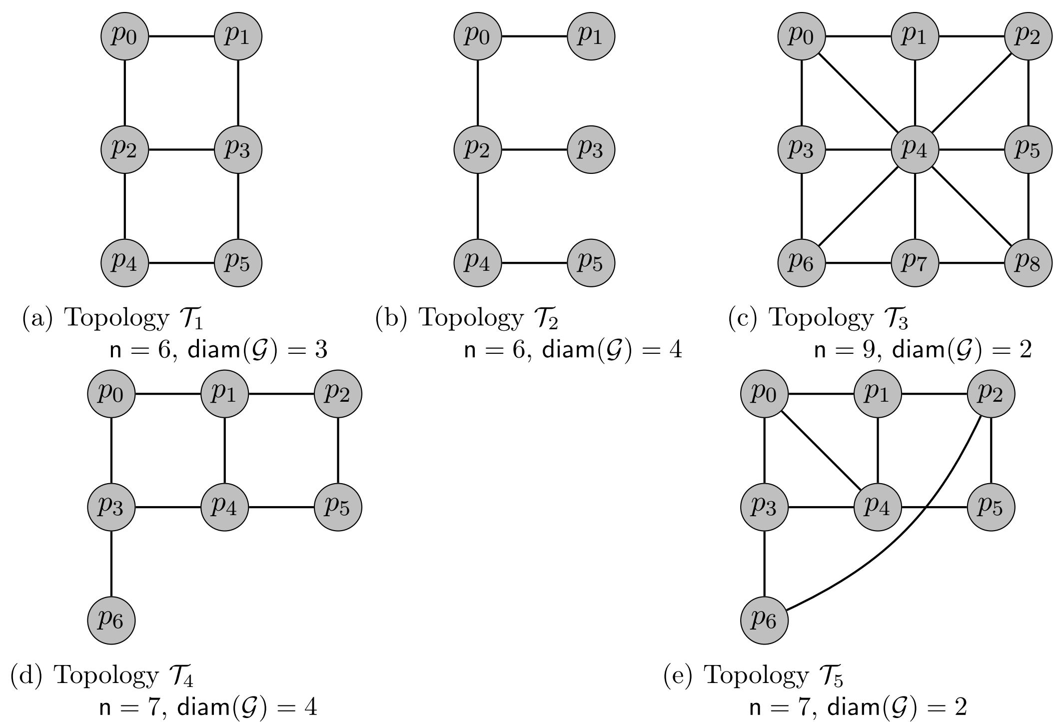
Recall that the specification of mutual exclusion comprises two properties: at most one process is privileged in each configuration, and each process is privileged infinitely often. The results of Chapter 4 state that Algorithm 4.2 is self-stabilizing wrt. mutual exclusion in $\lceil \mathsf{diam}(\mathcal{G})/2 \rceil - 1$ steps. This section concerns the recurrence of granting a privilege for processes beyond mutual exclusion. Recall that this amounts to analyzing the so-called *service time* [Joh02, Joh04].

Let the condition *priv* denote that at least one process is privileged – cf. Chapter 4. The recurrence properties to be considered are $priv_\Delta$- convergence and warmup.

The analysis tackles five graph topologies, given in Figure 7.1. The topologies have different combinations of the values of n (the number of processes) and $\mathsf{diam}(\mathcal{G})$ (the diameter of the topology), since n and $\mathsf{diam}(\mathcal{G})$ are the key parameters of the algorithm. Algorithm 4.2 is executed over these topologies, and the nuXmv model checker is used to analyze the $priv_\Delta$- convergence and warmup times for many values of Δ, which cover small and high recurrences for generality. Recall that Algorithm 4.2 achieves $priv_\Delta$, where $\Delta = \frac{\mathsf{n}}{\mathsf{n}+\lceil \mathsf{diam}(\mathcal{G})/2 \rceil - 1}$, with a $priv_\Delta$-convergence time complexity of $\max\{(\lceil 2.5 \cdot \mathsf{diam}(\mathcal{G}) \rceil - 1), (\mathsf{n} + \lceil \mathsf{diam}(\mathcal{G})/2 \rceil - 2)\}$.

The results are given in Table 7.1. Each row presents one test over the following elements given by the columns: the topology, the value of Δ, the property whether it is convergence or warmup, the convergence or warmup time, whether the property holds, and the testing time. The rows are grouped by the topology. The first and most important result is that the satisfiability of the properties adhere to the results of the formal proofs given in Chapter 4. Second, the testing time does not differ large when the value of Δ differs. It differs when the size n or the diameter $\mathsf{diam}(\mathcal{G})$ of the topology changes: the testing time increases when the size n increases (e.g. the difference between $\mathcal{T}_3$ and $\mathcal{T}_4$.) In addition, the testing time increases when the diameter increases (e.g. the difference between $\mathcal{T}_4$ and $\mathcal{T}_5$.)

To conclude, the presented approach aims to reduce the problem of verifying recurrence properties over infinite executions, to finite execution prefixes of a fixed length. It is shown that this approach makes the use of automatic verification – like model checking – much more efficient. A usefulness of this approach is shown by the example of the mutual exclusion algorithm.



(a) Topology $\mathcal{T}_1$
$\mathsf{n} = 6$, $\mathsf{diam}(\mathcal{G}) = 3$

(b) Topology $\mathcal{T}_2$
$\mathsf{n} = 6$, $\mathsf{diam}(\mathcal{G}) = 4$

(c) Topology $\mathcal{T}_3$
$\mathsf{n} = 9$, $\mathsf{diam}(\mathcal{G}) = 2$

(d) Topology $\mathcal{T}_4$
$\mathsf{n} = 7$, $\mathsf{diam}(\mathcal{G}) = 4$

(e) Topology $\mathcal{T}_5$
$\mathsf{n} = 7$, $\mathsf{diam}(\mathcal{G}) = 2$

Figure 7.1: Topologies, over which Algorithm 4.1 is tested

Topology	Δ	Property	#Steps	Holds	Testing Time
$\mathcal{T}_1$	$^6/_7$	$priv_\Delta$-convergence	$\mathsf{c} = 7$	✓	10s
$\mathcal{T}_1$	$^2/_{11}$	$priv_\Delta$-warmup	$\mathsf{w} = 9$	✓	14s
$\mathcal{T}_1$	$^4/_7$	$priv_\Delta$-warmup	$\mathsf{w} = 17$	✓	15s
$\mathcal{T}_1$	$^5/_7$	$priv_\Delta$-warmup	$\mathsf{w} = 37$	✓	29s
$\mathcal{T}_2$	$^6/_7$	$priv_\Delta$-convergence	$\mathsf{c} = 9$	✓	4s
$\mathcal{T}_2$	$^2/_{11}$	$priv_\Delta$-warmup	$\mathsf{w} = 9$	✓	6s
$\mathcal{T}_2$	$^4/_7$	$priv_\Delta$-warmup	$\mathsf{w} = 24$	✓	13s
$\mathcal{T}_2$	$^5/_7$	$priv_\Delta$-warmup	$\mathsf{w} = 51$	✓	19s
$\mathcal{T}_3$	$^1/_1$	$priv_\Delta$-convergence	$\mathsf{c} = 8$	✓	86h
$\mathcal{T}_3$	$^2/_{11}$	$priv_\Delta$-warmup	$\mathsf{w} = 9$	✓	73h
$\mathcal{T}_3$	$^4/_7$	$priv_\Delta$-warmup	$\mathsf{w} = 17$	✓	86h
$\mathcal{T}_3$	$^5/_7$	$priv_\Delta$-warmup	$\mathsf{w} = 37$	✓	88h
$\mathcal{T}_4$	$^7/_8$	$priv_\Delta$-convergence	$\mathsf{c} = 8$	X	5m51s
$\mathcal{T}_4$	$^7/_8$	$priv_\Delta$-convergence	$\mathsf{c} = 9$	✓	7m15s
$\mathcal{T}_4$	$^2/_{11}$	$priv_\Delta$-warmup	$\mathsf{w} = 8$	X	6m10s
$\mathcal{T}_4$	$^2/_{11}$	$priv_\Delta$-warmup	$\mathsf{w} = 9$	✓	7m55s
$\mathcal{T}_4$	$^4/_7$	$priv_\Delta$-warmup	$\mathsf{w} = 23$	X	9m42s
$\mathcal{T}_4$	$^4/_7$	$priv_\Delta$-warmup	$\mathsf{w} = 24$	✓	9m27s
$\mathcal{T}_4$	$^5/_7$	$priv_\Delta$-warmup	$\mathsf{w} = 43$	X	10m35s
$\mathcal{T}_4$	$^5/_7$	$priv_\Delta$-warmup	$\mathsf{w} = 44$	✓	10m44s
$\mathcal{T}_5$	$^1/_1$	$priv_\Delta$-convergence	$\mathsf{c} = 3$	X	3m03s
$\mathcal{T}_5$	$^1/_1$	$priv_\Delta$-convergence	$\mathsf{c} = 4$	✓	2m42s
$\mathcal{T}_5$	$^1/_1$	$priv_\Delta$-convergence	$\mathsf{c} = 5$	✓	2m42s
$\mathcal{T}_5$	$^1/_1$	$priv_\Delta$-convergence	$\mathsf{c} = 6$	✓	3m10s
$\mathcal{T}_5$	$^2/_{11}$	$priv_\Delta$-warmup	$\mathsf{w} = 2$	X	4m26s
$\mathcal{T}_5$	$^2/_{11}$	$priv_\Delta$-warmup	$\mathsf{w} = 3$	✓	3m24s
$\mathcal{T}_5$	$^4/_7$	$priv_\Delta$-warmup	$\mathsf{w} = 7$	X	4m05s
$\mathcal{T}_5$	$^4/_7$	$priv_\Delta$-warmup	$\mathsf{w} = 8$	✓	3m29s
$\mathcal{T}_5$	$^5/_7$	$priv_\Delta$-warmup	$\mathsf{w} = 11$	X	3m31s
$\mathcal{T}_5$	$^5/_7$	$priv_\Delta$-warmup	$\mathsf{w} = 12$	✓	4m

Table 7.1: Model checking recurrence properties for Algorithm 4.2 on a 64-core AMD Opteron with 2.6GHz, 504GiB of RAM (single-core mode)

8 Conclusion

This chapter concludes the thesis. It gives a summary of the presented work and some future prospects.

8.1 Summary

This work contributes to the area of self-stabilizing systems, by a new generalized concept of self-stabilization. The generalized concept comprises a new sort of properties, other than the classical safety properties defined over configurations. In this concept, the definition of a property is based on a measure that denotes the ratio at which a condition con is satisfied in a finite execution Ξ, which is signified as the recurrence of the condition in the execution $\mathsf{Rec}_{\mathsf{con}}(\Xi)$. The property is denoted by con_{Δ}, and is satisfied by an infinite execution Ξ if each execution prefix of Ξ guarantees a minimum recurrence Δ of con. Self-stabilization wrt. con_{Δ} reflects the convergence time required to achieve an execution suffix that satisfies con_{Δ} (con_{Δ}-convergence time). The classical self-stabilization is reflected in this concept by setting Δ to 1.0.

This generalized concept is also modelled for real-time systems: a property con_{Δ} is satisfied in an evolution $\mathcal{I}$ over a time interval $[t_1, \infty)$ iff in each interval $[t_1, t_2]$, the ratio of the accumulative time in which con is satisfied is greater than or equals Δ.

In contrast to classical self-stabilization, the condition con might be satisfied by a configuration during the convergence wrt. con_{Δ}, before reaching an execution suffix guaranteeing con_{Δ}. This provides a motivation to consider the satisfaction of con during the convergence. This issue is addressed by the new notion of con_{Δ}-warmup time: the time required to reach a configuration γ_i, such that the recurrence of con in $\gamma_0, \ldots, \gamma_j$ (i.e. from the beginning) where $j \geq i$ is greater than or equals Δ.

In this thesis, self-stabilization with recurrence properties is applied to solve problems in distributed computing. First, self-stabilization wrt. mutual exclusion is considered. The main issue that is analyzed is the recurrence of granting a privilege to an arbitrary process, after the corresponding algorithm stabilizes. This reflects analyzing the service time of the algorithm. The major contribution is a self-stabilizing mutual exclusion algorithm for general connected graphs under the synchronous scheduler, whose convergence time wrt. mutual exclusion is optimal ($\lceil \mathsf{diam}(\mathcal{G})/2 \rceil - 1$), and achieves $\Delta = 1.0$ recurrence (optimal) of granting a privilege in at most $\lceil 2.5 \cdot \mathsf{diam}(\mathcal{G}) - 1 \rceil$ steps. In addition, other algorithms showing the trade-off between convergence time wrt. mutual exclusion, the achieved recurrence of granting a privilege, and the space requirement are given.

Second, the problem of educated unique process selection is introduced. This problem is a generalized version of mutual exclusion, with a special consideration of the fairness property. Unique process selection holds if two properties are satisfied. The first property

indicates that at most one process is privileged in each configuration. The second property indicates that granting a privilege to an arbitrary process happens infinitely often, however, not necessarily for each process. The motivation of this problem is to direct the focus to increasing the recurrence of granting a privilege rather than mattering about fairness. This is useful in environments where processes rarely request a privilege, and serving them quickly implies satisfying fairness anyway. Educated unique process selection adds a restriction that if a process is granted a privilege, then this process is distinguished from all other processes based on some local or global criterion. Two self-stabilizing algorithms wrt. educated and unique process selection, respectively, are introduced. The algorithms are based on a Propagation of Information with Feedback (PIF) scheme for tree topologies.

The third problem is the slot assignment in Time Division Multiple Access (TDMA) protocols, given a tree topology and limited communication bandwidth. A formal analysis of the slot assignment order impact on the clock synchronization precision and the required length of the additional intervals – guard time – is given. The analysis specifies the slot assignment orders that yield the highest and lowest required guard times to guarantee message collision- and loss-free communication. This results in specifying tight lower bounds on the recurrence of sending messages for each slot assignment order. In addition, this work provides equations to compute safe and optimal guard times for many cases. A case study of a wireless fire alarm system is provided. The study shows the usefulness of the given approach to optimize the guard time length, and therefore, to increase the recurrence of sending messages in a real-world system.

Finally, an automatic verification approach for verifying recurrence properties is provided. The approach basically tackles the issue of having recurrence properties being defined over infinite executions, which complicates the verification process. The approach makes use of having each configuration initial, to reduce the problem to verifying whether there exists an execution prefix having some fixed length, and violates con_Δ-convergence of c steps or con_Δ-warmup of w steps. For the former case, the length is $\mathsf{c}+1$, and for the latter case, the length is in $[\mathsf{w}+1, 2\mathsf{w}+1]$. This approach is shown to be useful, by being applied to verify recurrence properties of a mutual exclusion algorithm executed over some topologies.

The whole contribution of this thesis is sketched in Figure 8.1.

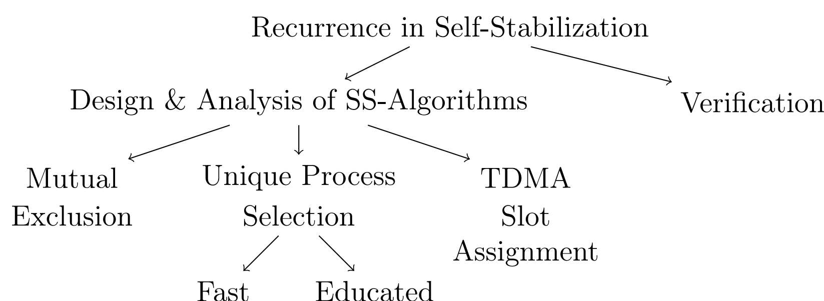


Figure 8.1: Wrap-up

The following points summarize how this contribution would impact self-stabilization:

- Usually, when self-stabilization is considered for properties defined over executions, it captures only specific scenarios, because self-stabilization is mostly considered for properties defined over configurations. The formalism of self-stabilization with recurrence properties provides some pattern for modelling properties defined over executions. As shown in the previous chapters, this pattern covers many aspects describing executions.

- Optimizing the convergence time and the space requirement for self-stabilizing solutions wrt. mutual exclusion and slot assignment usually neglects the relation between the convergence time and the achieved performance or quality of service. For the two problems, the generalized concept has highlighted the convergence wrt. the desired service time in mutual exclusion, and the desired message delivery (throughput) with slot assignment. In addition, the unique and educated process selection highlight the service time issue further, by adding assumptions over the fairness properties and the environment.

- While automatic verification of properties over distributed algorithms is hard, the verification approach, presented in Chapter 7, has added another methodology to simplify automatic verification of some sort of properties in self-stabilizing distributed algorithms. In addition, the case study in Chapter 7 has shown the usefulness of the model checker nuXmv to verify this sort of properties in this environment.

8.2 Future Prospects

Some future prospects are given in this section.

The generalized concept of self-stabilization covers recurrence properties for both discrete and real-time systems. Recurrence is, roughly, a ratio defined over subsequent configurations and time intervals, respectively. This ratio is required to hold for any subexecution or interval, starting from some point. Two limitations with ideas for improvement are pointed out for the definition of recurrence.

- By definition, in order that the recurrence property con_Δ holds in an execution, the distribution of the configurations that satisfy the condition con over any prefix has to somehow be uniform, or Δ should be very small. In other words, the corresponding condition has to keep repeating without exceeding a latency margin, in order to preserve Δ for each prefix. This is, for example, not the case for Algorithm 5.6. In order to overcome this limitation, the definition can be generalized further to cover that a condition is required to hold at least once in segments of executions or evolutions, rather than configurations.

- Recurrence is a ratio at which a condition holds globally. For distributed systems, a particular condition defined over states may hold for one or more states, providing better or worse performance. The local mutual exclusion problem is an example of this aspect, where multiple processes may be granted a privilege under some restrictions. This cannot be modelled by the given definition. This limitation can

be overcome by introducing a weight reflecting the number of states at which the condition holds. This weight can be involved in the computation of Δ.

In the presented work, recurrence properties focus on modelling performance aspects in self-stabilization. Following the same definition, instead of modelling performance related conditions, one may focus on the consequences of faults during the convergence of a self-stabilizing system. In particular, the notion of warmup time can be used to analyze the recurrence of conditions that are followed by non-desired actions during convergence. For example, the condition may be defined to reflect *failing to satisfy mutual exclusion*; i.e. the condition holds if two or more processes are granted a privilege in the same configuration. This provides more information about the algorithm's behavior during convergence, which might help to improve the quality of service.

Regarding the mutual exclusion and educated unique process selection problems (cf. Chapters 4 and 5), a future perspective can be optimizing the space requirement and generalizing the scheduler for the given solutions. For example, one may study the feasibility of designing an asynchronous unison without having particular values for the reset command, as given in the state-of-the-art [BPV04]. This reduces not only the space requirement, but also the convergence time wrt. the desired recurrence of granting a privilege.

Concerning the verification approach given in Chapter 7, systems are assumed to have a finite configuration space, and model checking is applied directly. For future work, systems with infinite configuration's space can be considered. The infinite configuration space is usually the result of having variables with infinite domains. Such variables mostly have number values. This raises the question whether bounded model checking (BMC) [Str00, CBRZ01] together with abstraction techniques are useful to verify recurrence properties for these systems. BMC is based on checking whether executions having an upper bound on their lengths satisfy some property, by translating the executions to formulas that are solved by SAT solvers [DP60, DLL62] and SMT solvers [dMB11]. From this point, BMC is supposed to be useful for extending this work to consider systems having an infinite configuration space.

A Correctness Proofs for Chapter 5

A.1 Proof of Theorem 5.1

Lemma A.1. Given an execution step (γ_{i-1}, γ_i) of a topology of depth d, if $\gamma_{i-1} \models \mathsf{root}_{\bot}^{\top}$ and $\gamma_i \models \mathsf{root}_{\top}^{\top}$, then the configuration $\gamma_{i+\mathsf{depth}(\mathcal{T})}$ is a legConfig.

Proof. By Algorithm 5.5, the root executes either gc_3 or gc_4 in the step, implying that in γ_i, each root's child ch is in a state $ch_{\bot}^{\bot}$. In (γ_i, γ_{i+1}), each ch executes one of $\mathsf{gc}_5, \mathsf{gc}_6, \mathsf{gc}_{13}$ or gc_{14}, and each process q in depth 2 is in a state $q_{\bot}^{?}$ in configuration γ_{i+1}, following that $ch_{\bot}^{\bot}$ in γ_i. By Definition 5.7, in γ_{i+1}, the states of processes in the 1-subtree satisfy the conditions of legConfigs, if the fact that $\mathsf{up} = \bot$ for the leaves is ignored. Note that the processes at depth 2 do not create bottom tokens to the processes in depth 1. Inductively, in the next steps, in each configuration γ_j for $i+2 \leq j \leq i+d$, the states of processes in the j-subtree satisfy the conditions of legConfig's, implying that a legConfig is reached in d steps (in γ_{i+d}). □

Lemma A.2. Given an execution step (γ_{s-1}, γ_s) of a topology of depth d, if $\gamma_{s-1} \models \mathsf{root}_{\top}^{\top}$ and $\gamma_s \models \mathsf{root}_{\bot}^{\top}$, then the configuration γ_{s+d} is a legConfig.

Proof. In any step (γ_{s-1}, γ_s), if $\gamma_{s-1} \models \mathsf{root}_{\top}^{\top}$ and $\gamma_i \models \mathsf{root}_{\bot}^{\top}$, then each root's child ch is in a state $ch_{\top}^{?}$ in γ_s, and the root executes gc_1 or gc_3 in the step. The root can point to at most one of its children. In the step (γ_s, γ_{s+1}), each root's child ch switches into $ch_{\bot}^{\bot}$ except at most one of them (switches into $ch_{\bot}^{\top}$) and only if there is an *ePath*. Note that the 1-subtree satisfy legConfig conditions in γ_{s+1}, except at most the condition (5.6): if there is an *ePath*, it might not end with an active process. The processes at depth 2 do not create bottom tokens to the processes in depth 1. Inductively, in the next steps, if the root does not change its state, then in each γ_j for $(i+2) \leq j \leq (d-1)$, the states of processes in the j-subtree satisfy legConfig conditions, except at most the condition (5.6). In γ_d, the tree satisfies legConfig, including the condition (5.6). If the root switches into $\mathsf{root}_{\top}^{\top}$ during these steps, by Lemma A.1, the token sent from the root does not violate reaching a legConfig in d steps. □

Proof of Theorem 5.1

Proof. By Lemma 5.3, for each execution, the root changes its state from $\mathsf{root}_{b}^{\top}$ into $\mathsf{root}_{\neg b}^{\top}$ in $2d$ steps. By Lemmata A.1 and A.2, a legConfig is reached in d steps after the root changes its state. The sum is equal to $d + 2d = 3d$. □

A.2 Proof of Theorem 5.2

Definition A.1. Given the legConfigs' space $\Gamma_{leg} \subset \Gamma$, four subsets of configurations *C1*, *C2*, *C3*, *C4*, where $C1 \dot{\cup} C2 \dot{\cup} C3 \dot{\cup} C4 \subseteq \Gamma_{leg}$, are defined as follows:

- $C1 ::= \{\gamma \in \Gamma_{leg} \mid \gamma \models \mathsf{root}^{\top}_{\top}\}$
- $C2 ::= \{\gamma \in \Gamma_{leg} \mid \gamma \models \exists\, ePath_{\gamma} : \mathsf{root}, \ldots, p_s \bullet p_s{}^{?}_{\top}\}$
- $C3 ::= \{\gamma \in \Gamma_{leg} \mid \gamma \models \exists\, ePath_{\gamma} : \mathsf{root}, \ldots, p_s \bullet p_s{}^{?}_{\bot}\}$
- $C4 ::= \{\gamma \in \Gamma_{leg} \mid \gamma \models \mathsf{root}^{\top}_{\bot} \wedge \nexists\, p \in \mathcal{P} \setminus \{\mathsf{root}\} \bullet p.l\text{-}active\}$ ♢

Corollary A.1. $\Gamma_{leg} = C1 \dot{\cup} C2 \dot{\cup} C3 \dot{\cup} C4$

The following lemma concerns the closure of configurations satisfying Properties 1 and 2 in Definition 5.7.

Lemma A.3. Given an execution $\gamma_0, \gamma_1, \ldots$ over a topology, if $\gamma_0 \in \Gamma_{leg}$, then for each configuration γ_i, where $i \geq 0$:

$$\forall\, p \in \mathcal{P} \bullet (p^{\top}_{\top} \longrightarrow p^{\top}_{\top}\text{-}1 \wedge (p \neq \mathsf{root} \longrightarrow \neg p.l\text{-}active)) \tag{A.1}$$

$$\forall\, p_s \in \mathcal{P} \bullet p_s{}^{\bot}_{\top} \longrightarrow$$

$$\exists\, r \geq s \bullet p_s, \ldots, p_r : aPath \vee \tag{A.2}$$

$$\forall\, path : p_s, \ldots, p_z \bullet \forall\, e \in [s, z] \bullet \neg p_e.l\text{-}active \wedge p_e{}^{\bot}_{\top}\text{-}1 \tag{A.3}$$

Proof. It is shown that the conditions (A.1)–(A.3), reflecting the conditions (5.1)–(5.3) in Definition 5.7, hold in each γ_i by induction. For the base case γ_0, the conditions hold by construction. For the induction step, it is shown that if the conditions hold in a configuration γ_{j-1}, then they hold in γ_j for $j \in \mathbb{N}_0$. Regarding the condition (A.1): By hypothesis, in γ_{j-1}, for each process p where $p^{\top}_{\top}$, it holds that $p.\ell = -1 \wedge \neg p.l\text{-}active$. By Algorithm 5.5, the value of ℓ changes in γ_j only if: (i) a process $p^{\top}_{\bot}$ switches into $p^{\bot}_{\bot}$, or (ii) a process $p^{?}_{\bot}$ switches into $p^{\top}_{\top}\text{-}1$ or $p^{\bot}_{\top} \neq 1$. In both cases, $\gamma_j : p^{\top}_{\top} \longrightarrow p^{\top}_{\top}\text{-}1$. If a non-root process p is active, this can be last checked only in Algorithm 5.5, by gc_6 and gc_{14}. In both actions, p switches into $p^{\bot}_{\top}$, but not into $p^{\top}_{\top}$. Thus, the condition (A.1) holds. Regarding the conditions (A.2) and (A.3): By hypothesis, the condition holds in γ_{i-1}. By definition, always $\mathsf{root.up} = \top$. Therefore, the condition holds trivially for the root. For a non-root process p where $\gamma_{j-1} : \neg(p^{\bot}_{\top})$, p switches into $p^{\bot}_{\top}$ in γ_j iff any of the following commands is enabled:

- ($\mathsf{gc}_6, \mathsf{gc}_{14}$) p switches into $p^{\bot}_{\top}\mathsf{id}$ in γ_j. Note that γ_j has an active path with a process p satisfying the sub-condition (A.2).
- (gc_7) p switches into $p^{\bot}_{\top}ch.\mathsf{id}$ forming an active path $p, ch, \ldots, p_s$ for $p_s \in \mathcal{P}$, satisfying the sub-condition (A.2).
- (gc_8) p switches into state $p^{\bot}_{\top}\text{-}1$, which satisfies the sub-condition (A.3). □

The following lemmata concern execution steps' properties within the four categories.

Lemma A.4. For each execution prefix $\gamma_0, \ldots, \gamma_s$ over a topology, if $\gamma_0 \in C1$ and $\gamma_i \models \mathsf{root}_{\top}^{\top}$ for $0 \le i \le s$, then $\gamma_i \in C1$.

Proof. Given that the state $\mathsf{root}_{\top}^{\top}$ holds in γ_i, the statement $\gamma_i \in C1$ holds if the conditions (5.1)–(5.5) in Definition 5.7 hold. The conditions (5.1)–(5.3) hold in γ_i by Lemma A.3. It is shown that the conditions (5.4) and (5.5) hold by induction: the base case γ_0 holds by construction. For the induction step: it is shown that if the conditions hold in a configuration γ_{j-1}, then they hold in γ_j for $j \in \mathbb{N}_0$. Regarding the condition (5.4), a process p switches into $p_{\bot}^{\top}$ only if gc_{10} is enabled. gc_{10} cannot be enabled because the condition (5.5) holds in γ_{j-1} and contradicts the guard of gc_{10}. In addition, if a process q switches into $q_{\top}^{\top}$, then the value of ℓ is always -1 (gc_{3-5}). This implies that the condition (5.4) holds in γ_j. Regarding the condition (5.5): Given that the conditions (5.4) and (5.5) hold in γ_{j-1} by hypothesis, there is no process q in a state $q_{\bot}^{?}$ and points to one of its children. From this point, if a process p switches into $p_{\bot}^{?}$, it does not point to any of its children. This implies that the condition (5.5) holds in γ_j. □

Lemma A.5. Given a legConfig $\gamma_0 \in C1$ of a topology $\mathcal{T}$, for each execution $\gamma_0, \gamma_1, \ldots,$ there exists finally a configuration γ_s, where $s > 0$, where: (1) if there exists at least one last-active non-root process in γ_{s-1}, then $\gamma_s \in C2$ and $s \le d$, where d is the minimum depth of a last-active process, and (2) $\gamma_s \in C4$ and $s \le \mathsf{depth}(\mathcal{T})$ if there is no last-active non-root process in γ_{s-1}, such that $\forall\, 0 \le i \le s-1 \bullet \gamma_i \in C1$.

Proof. By Lemma A.4, if $\mathsf{root}_{\top}^{\top}$ holds in γ_i, then $\gamma_i \in C1$. By Lemma 5.3, the root switches into $\gamma_s : \mathsf{root}_{\bot}^{\top}$ in at most $2\mathsf{depth}(\mathcal{T})$ steps, by executing one of the commands gc_1 or gc_2: (a) If there is a last-active process in γ_{s-1}, then by the condition (5.2) in Definition 5.7, there exists an $aPath_{\gamma_{s-1}}$ implying that the root executes gc_1, and $\gamma_s \in C2$. Note that by Lemma 5.2, if there is an *aPath* having a last-active process with the minimum depth d, then $s = 2d$. (b) If there is no *active* process in γ_{s-1}, then the condition (5.3) in Definition 5.7 holds in γ_{s-1}, implying that the root executes gc_2, and $\gamma_s \in C4$. □

Lemma A.6. For each execution prefix $\gamma_0, \ldots, \gamma_s$ over a topology $\mathcal{T}$, if $\gamma_0 \in C2 \cup C3$ and $\gamma_i \models \mathsf{root}_{\bot}^{\top}$ for $0 \le i \le s$, then $\gamma_i \in C2 \cup C3$.

Proof. Given that $\mathsf{root}_{\bot}^{\top}$ holds in γ_i, it follows that $\gamma_i \in C2 \cup C3$ if the conditions (5.1)–(5.3), and (5.6) in Definition 5.7 hold. The conditions (5.1)–(5.3) hold in γ_i by Lemma A.3. The condition (5.6) holds by induction (analogous to proving Lemmata A.3 and A.4). □

Lemma A.7. Given a legConfig $\gamma_0 \in C2$ of a topology $\mathcal{T}$, for each execution $\gamma_0, \gamma_1, \ldots,$ there exists a configuration γ_u for $u > 0$, such that

- $\gamma_u \in C3$ and $\forall\, 0 \le i \le u-1 \bullet \gamma_i \in C2$, and
- for the step (γ_{u-1}, γ_u), there is exactly one process $p \neq \mathsf{root}$ such that $\gamma_{u-1} : p_{\top}^{\bot} p.\mathsf{id}$, $\gamma_u : p_{\bot}^{\bot} p.\mathsf{id}$, $u = \mathsf{depth}(p)$, and p executes the command *critSection*() in the step.

Proof. By Lemma A.6, if the state $\mathsf{root}_{\bot}^{\top}$ holds in γ_i, then $\gamma_i \in C2 \cup C3$. In general, given a legConfig $\gamma \in C2 \cup C3$ and an *ePath* : $p_0, \ldots, p_s$: it holds that $(p_s{}_{\top}^{\bot} \longrightarrow \gamma \in C2) \wedge (p_s{}_{\bot}^{\bot} \longrightarrow \gamma \in C3)$. Given that γ_0 is in $C2$, then for the $ePath_{\gamma_0}$: $p_0, \ldots, p_s$, it holds that $\gamma_0 : p_s{}_{\top}^{\bot} p_s$. By Definition 5.6, the $ePath_{\gamma}$ can be expressed as follows:

$$p_0{}_{\bot}^{\top} p_1, p_1{}_{\bot}^{\top} p_2, \ldots, p_k{}_{\bot}^{\top} p_{j+1}, p_{j+1}{}_{\top}^{\bot} p_{j+2}, \ldots, p_s{}_{\top}^{\bot} p_s \tag{A.4}$$

By Algorithm 5.5 and Lemma 5.2: The processes $p_0, \ldots, p_j,\ p_{j+2}, \ldots, p_s$ have no top tokens. The processes $p_0, \ldots, p_s$ have no bottom tokens. The only process with a token is p_{j+1} (gc_{10}), and it is a top token. In γ_1, p_{j+1} switches into $p_{j+1}{}_{\bot}^{\top} p_{j+2}$. Note that $\gamma_1 \in C2$. Analogously, p_{j+2} performs the same action in γ_2. Since the number of $p_{j+1}, \ldots, p_s$ is up to $\mathsf{depth}(p_s) - 1$, then a configuration γ_{u-1} is reached where $u \leq \mathsf{depth}(p_s)$ and

$$p_0{}_{\bot}^{\top} p_1, p_1{}_{\bot}^{\top} p_2, \ldots, p_{s-1}{}_{\bot}^{\top} p_s, p_s{}_{\top}^{\bot} p_s \tag{A.5}$$

In γ_{u-1}, by Algorithm 5.5, the only process with a token is p_s (gc_9) which is a top token. In the execution step (γ_{u-1}, γ_u), p_s executes *critSection*() and switches into $p_s{}_{\bot}^{\top} p_s$, implying that $\gamma_u \in C3$. □

Lemma A.8. Given a legConfig $\gamma_0 \in C3$ of a topology $\mathcal{T}$, and an *ePath* : $p_0, \ldots, p_s$, for each execution $\gamma_0, \gamma_1, \ldots$, there exists a configuration γ_u for $0 < u \leq \mathsf{depth}(p_s)$, such that $\gamma_u \in C1$ and $\forall\, 0 \leq i \leq u - 1 \bullet \gamma_i \in C3$.

Proof. By Lemma A.6, if $\mathsf{root}_{\bot}^{\top}$ in γ_i, then $\gamma_i \in C2 \cup C3$. In configuration $\gamma_0 \in C3$, the $ePath_{\gamma_0}$ can be expressed as follows:

$$p_0{}_{\bot}^{\top} p_1, p_1{}_{\bot}^{\top} p_2, \ldots, p_k{}_{\bot}^{\top} p_{j+1}, p_{j+1}{}_{\bot}^{\bot} p_{j+2}, \ldots, p_s{}_{\bot}^{\bot} p_s \tag{A.6}$$

By Lemma 5.2, the processes $p_0, \ldots, p_s$ have no top tokens, the processes $p_0, \ldots, p_{j-1}$, $p_{j+1}, \ldots, p_s$ have no bottom tokens, and by Algorithm 5.5, the only process which has a token is p_j (gc_{12}), and it is a bottom token. In the execution step (γ_0, γ_1), p_u switches into $p_s{}_{\bot}^{\bot} p_{j+1}$. Analogously, p_{j-1} performs the same step. Since the number of processes $p_1, \ldots, p_j$ is up to $\mathsf{depth}(p_s) - 1$, then in a configuration γ_{u-1} for $0 \leq u \leq \mathsf{depth}(p_s)$, the *ePath* is expressed as follows:

$$p_0{}_{\bot}^{\top} p_1, p_1{}_{\bot}^{\bot} p_2, \ldots, p_u{}_{\bot}^{\bot} p_{j+1}, p_{j+1}{}_{\bot}^{\bot} p_{j+2}, \ldots, p_s{}_{\bot}^{\bot} p_s \tag{A.7}$$

Note that in any step following that $\mathsf{root}_{\bot}^{\top} p_1$, each root child *ch* other than p_1 is in a state $ch_{\bot}^{\bot}$. This implies that in γ_{u-1}, the root has a bottom token. In the step (γ_{u-1}, γ_u), the root switches into a configuration $\gamma_u \in C1$ where $\mathsf{root}_{\top}^{\top}$-1 ($\mathsf{gc}_3, \mathsf{gc}_4$), after executing *critSection*() if it is active in γ_{u-1} (gc_3). □

Lemma A.9. For each execution prefix $\gamma_0, \ldots, \gamma_s$ over a topology $\mathcal{T}$, if $\gamma_0 \in C4$ and $\gamma_i : \mathsf{root}_{\bot}^{\top}$ for $0 \leq i \leq s$, then $\gamma_i \in C4$.

Proof. Given that the state $\mathsf{root}_{\top}^{\top}$ holds in γ_i, it follows that $\gamma_i \in C4$ if the conditions (5.1)–(5.3), (5.7)–(5.8) in Definition 5.7 hold. The conditions (5.1)–(5.3) hold in γ_i by Lemma A.3. The conditions (5.7) and (5.8) hold by induction (analogous to proving Lemmata A.3 and A.6). □

Lemma A.10. Given a legConfig $\gamma_0 \in C4$ of a topology $\mathcal{T}$, for each execution $\gamma_0, \gamma_1, \ldots$, there exists a configuration γ_u for $0 < u \leq 2$, such that $\gamma_u \in C1$ and $\forall\, 0 \leq i \leq u-1 \bullet \gamma_i \in C4$.

Proof. By Lemma A.9, if $\gamma_0 \in C4$ and $\mathsf{root}^{\top}_{\perp}$ holds in γ_i, then $\gamma_i \in C4$. If each root's child ch is in a state $ch^{\perp}_{\perp}$, the root switches into $\mathsf{root}^{\top}_{\top}$ in $\gamma_1 \in C1$. Otherwise, by Algorithm 5.5-gc_{11}, each root's child switches into $ch^{\perp}_{\perp}$-1 in γ_1. In γ_2, by gc_3 or gc_4, the root switch into $\mathsf{root}^{\top}_{\top}$-1 such that $\gamma_2 \in C1$. □

Proof of Theorem 5.2

Proof. If $\gamma_0 \in \Gamma_{leg}$, then by Corollary A.1, γ_0 is in one of the categories $C1$, $C2$, $C3$, and $C4$:

- If $\gamma_0 \in C1$, then by Lemmata A.4 and A.5, γ_1 is in one of $C1$, $C2$, and $C4$.
- If $\gamma_0 \in C2$, then by Lemmata A.6 and A.7, γ_1 is in one of $C2$ and $C3$.
- If $\gamma_0 \in C3$, then by Lemmata A.6 and A.8, γ_1 is in one of $C3$ and $C1$.
- If $\gamma_0 \in C4$, then by Lemmata A.9 and A.10, γ_1 is in one of $C4$ and $C1$.

This implies the closure of the set of legConfigs. □

A.3 Proof of Theorem 5.3

Lemma A.11. Unique process selection is satisfied in each legConfig.

Proof. The commands, whose actions comprise *critSection*(), are $\mathsf{gc}_3, \mathsf{gc}_9$, and gc_{15}. It is shown that in a legConfig, if one is enabled, then the others are not.

1. If gc_3 is enabled, then each root's child ch is in a state $ch^{\perp}_{\perp}$. (a) If there exists an *ePath* : $p_0, \ldots, p_u$, then by Definition 5.6, $p_j{}^{\perp}_{\perp}$ for $1 \leq j \leq u$. This implies that gc_9, gc_{12} are not enabled for p_j. For each process q that is not in the *ePath*, the following holds:

$$(q^{\top}_{?} \longrightarrow q^{\top}_{\top}\text{-}1) \wedge (q^{?}_{\top} \wedge \mathsf{parent}(q)^{?}_{\perp} \longrightarrow \mathsf{parent}(q).\ell \neq q.\mathsf{id}) \quad \text{(A.8)}$$

   By the condition (A.8), the commands gc_9 and gc_{12} are not enabled. (b) If there is no *ePath*, then the condition (A.8) holds, implying that gc_9 and gc_{12} are not enabled.

2. If any of gc_9 and gc_{12} is enabled for a process p_s, then $p_s{}^{\perp}_{\top}{}_{p_s.\mathsf{id}}$ holds, and by Definition 5.7, there exists an *ePath* : $p_0, p_1 \ldots, p_s$ such that only p_s enables gc_9 or gc_{12}. By (A.8), the processes that are not in the path do not enable gc_9 or gc_{12}. Regarding the root ($= p_0$), by Definition 5.6, p_1 is not in a state $p_1{}^{\perp}_{\perp}$, and therefore, gc_3 is not enabled. Consequently, *UPS* holds in each legConfig. □

Proof of Theorem 5.3

PROOF. By Lemma A.11, *UPS* holds in each legConfig. By Theorem 5.2, the set of legConfigs is closed under Algorithm 5.5. A non-root process p runs *critSection*() only if $\mathsf{parent}(p).\mathsf{x} \neq p.\mathsf{x}$ $(\mathsf{gc}_9, \mathsf{gc}_{15})$. The root executes *critSection*() only if each root's child ch is in a state $ch^{\perp}_{\perp}$. Consider three scenarios: (i) If the root does not change its state within $\gamma_0, \ldots, \gamma_d$, then by Lemma 5.2, all processes at depth $b \leq d$ will have the same value of x in a configuration γ_b, implying that no non-root process in $\mathcal{T}$ may execute *critSection*() in γ_d and any following step until the root changes its state into $\mathsf{root}^{\top}_{\perp}$ sending token_3. (ii) In any execution step (γ_{i-1}, γ_i) if $\gamma_{i-1} \models \mathsf{root}^{\top}_{\perp}$ and $\gamma_i \models \mathsf{root}^{\top}_{\top}$, the token created by the root does not enable a non-root process to execute *critSection*(). (iii) In any execution step (γ_{s-1}, γ_s), if $\gamma_{s-1} \models \mathsf{root}^{\top}_{\top}$ and $\gamma_i \models \mathsf{root}^{\top}_{\perp}$, then each root's child ch is in a state $ch^{?}_{\top}$ in γ_s. The root points to at most one of its children. In (γ_s, γ_{s+1}), each ch switches into $ch^{\perp}_{\perp}$ except at most one of them and only if there is an *ePath*. Now the 1-subtree satisfies unique process selection. Inductively, in the next steps, in each configuration γ_j for $s + 2 \leq j \leq d$, the j-subtree satisfies unique process selection.
Scenarios (i-ii) guarantee that no non-root process at depth $b \leq d$ executes *critSection*() after b steps unless the root switches into $\mathsf{root}^{\top}_{\perp}$. If the root switches into $\mathsf{root}^{\top}_{\perp}$ in γ_b, by Scenario (iii), for each configuration γ_r where $b \leq r \leq d$, the r-subtree satisfies unique process selection, while the processes whose depth is greater than r require $d - r$ steps to guarantee no execution of *critSection*(). This implies that unique process selection is achieved in $\gamma_d, \gamma_{d+1}, \ldots$. □

A.4 Proof of Theorem 5.4

LEMMA A.12. Let γ_0 be a legConfig in which *priv* holds. For each execution $\gamma_0, \gamma_1, \ldots$, there exists $i \leq 4 \cdot \mathsf{depth}(\mathcal{T})$ such that *priv* holds in γ_i.

PROOF. Let d be $\mathsf{depth}(\mathcal{T})$. By Theorem 5.2, any configuration in the execution is a legConfig. Since *priv* holds in γ_0, then there exists a last-active process p that is uniquely privileged. In the following scenarios, it is shown that a process is granted a privilege in $4d$ steps.

1. If p is the root, then by Definition A.1, $\gamma_0 \in$ *C3* $\cup$ *C4* and by Lemmata A.8 and A.10, $\gamma_1 \in$ *C1* holds. Next,
   a) If there is an active non-root process, by Lemmata A.5 and A.7, the process is granted a privilege in $3d{-}1$ steps – until γ_{3d}.
   b) If only the root is active in γ_1, by Lemmata A.5 and A.10, the root is granted a privilege in $2d$ steps – until γ_{2d+1}.
2. If p is not the root, then by Lemma A.7, $\gamma_0 \in$ *C2* and $\gamma_1 \in$ *C3*. Next,
   a) If the root is not last-active, then by Lemma A.8, after d steps, it holds that $\gamma_{d+1} \in$ *C1*. Following the argument of scenario $(1) - (a)$, a process is privileged until γ_{4d}.

b) If the root is last-active, then in $d-1$ steps, the root is granted a privilege – until γ_d.

Given by assumption that $d \geq 1$, by the above four scenarios, a process is granted a privilege in $4d$ steps. □

Proof of Theorem 5.4

Proof. Let d be $\mathsf{depth}(\mathcal{T})$. By Theorem 5.1, Algorithm 5.5 is self-stabilizing wrt. a legConfig in $3d$ steps, and is self-stabilizing wrt. *UPS* in d steps. By Lemmata A.5, A.7, A.8, and A.10, a process is granted a privilege in $4d$ steps after the algorithm stabilizes wrt. legConfig, i.e. until γ_{7d}. By Lemma A.12. Let γ_i be a configuration in which *priv* holds and $3d \leq i \leq 7d$. The recurrence of *priv* in any execution $\gamma_i, \gamma_{i+1}, \ldots$ is greater or equal to Δ. This implies that Algorithm 5.5 is self-stabilizing wrt. $priv_{\Delta}$ in $7d$ steps. □

A.5 Proof of Theorem 5.6

Definition A.2. Given the legConfigs' space $\Gamma_{leg} \subset \Gamma$, four subsets of configurations $C1$, $C2$, $C3$, and $C4$, where $C1 \mathbin{\dot\cup} C2 \mathbin{\dot\cup} C3 \mathbin{\dot\cup} C4 \subseteq \Gamma_{leg}$, are defined as follows:

- $C1 ::= \{\gamma \in \Gamma_{leg} \mid \gamma \models \mathsf{root}^{\top}_{\top}\}$
- $C2 ::= \{\gamma \in \Gamma_{leg} \mid \gamma \models \mathsf{root}^{\top}_{\bot}$ and there exists a path $\mathsf{root} = p_0, \ldots, p_s$ such that $p_s{}^{\bot}_{\top}\mathrm{s}$, and $\forall\, 0 \leq i < s \bullet p_i.\ell = i+1\}$
- $C3 ::= \{\gamma \in \Gamma_{leg} \mid \gamma \models \mathsf{root}^{\top}_{\bot}$ and there exists a path $\mathsf{root} = p_0, \ldots, p_s$ such that $p_s{}^{\top}_{\bot}\mathrm{s} \vee p_s{}^{\bot}_{\bot}\mathrm{s}$, and $\forall\, 0 \leq i < s \bullet p_i.\ell = i+1\}$
- $C4 ::= \{\gamma \in \Gamma_{leg} \mid \gamma \models \mathsf{root}^{\top}_{\bot} \wedge \mathsf{root}.\ell = \mathsf{root.id}\}$ ♢

Lemma A.13. $\Gamma_{leg} = C1 \mathbin{\dot\cup} C2 \mathbin{\dot\cup} C3 \mathbin{\dot\cup} C4$.

Proof. In general, in any legConfig, the root can be in state $\mathsf{root}^{\top}_{\top}$ or $\mathsf{root}^{\top}_{\bot}$. The former case is covered by $C1$. Next, it is shown that any legConfig γ with $\mathsf{root}^{\top}_{\bot}$ is in one of $C2, C3, C4$. By Definition 5.9, Property 4, if the root points to itself, then $\gamma \in C4$. Otherwise, there exists a path $\mathsf{root} = p_0, \ldots, p_s$ such that for all $0 \leq i < s$, $p_i.\ell = i+1$ and $p_s.\ell = s$. By Definition 5.9, Property 1, p_s cannot be in state $p_s{}^{\top}_{\top}$. If $p_s{}^{\bot}_{\top}$ holds, then $\gamma \in C2$. If $\varphi_s{}^{\top}_{\bot}$ or $p_s{}^{\bot}_{\bot}$ holds, then $\gamma \in C3$. □

Lemma A.14. Given a legConfig γ_0 in $C1$, for any step (γ_0, γ_1), if $\gamma_1 \models \mathsf{root}^{\top}_{\top}$, then $\gamma_1 \in C1$.

Proof. By definition of the commands, when a process changes its state, it changes at least one of the values of x and up. Note that in Definition 5.9, the pre-conditions of Properties 1–4 are related to the values of x and up. Therefore, it is basically referred to the values of x and up to show that $\gamma_1 \in C1$. By Definition A.2, since $\gamma_0 \in C1$, then $\mathsf{root}^{\top}_{\top}$ holds in γ_0. In the following, it is shown that each property in Definition 5.9 holds in γ_1.

Property 1: Since Property 1 holds in γ_0, then there is no process p in state $p^{\top}_{\bot}$ in γ_0. Whenever a process p_i switches from ${p_i}^{\bot}_{?}$ to ${p_i}^{\top}_{?}$, its is due to token_1 (Commands gc_4 and gc_{10}). By Property 1-a,b, if a process has a top token then, ${p_i}^{?}_{\bot}$ and $\mathsf{parent}(p_i)^{?}_{\top}$, implying that p_i switches the value of x into $\top$ in γ_1, and Property 1-a holds. Similarly, whenever a process p_j switches from ${p_j}^{\top}_{?}$ to ${p_j}^{\bot}_{?}$, then it is due to bottom token, implying that p_j does not change the value of x. This implies that Property 1-b holds in γ_1. *Property 2*: When a non-root process p changes its state into $p^{\top}_{\top}$, then it is the result of the command gc_4. The command sets m to $update_{\mathsf{m}}()$, implying that $\mathsf{m} = \mu$ in γ_1. Property 2 holds. *Property 3*: When a non-root process p switches into $p^{\bot}_{\top}$, then it is by either gc_5 or gc_{10}, and both of them result in a state satisfying Property 3. *Property 4* holds trivially by the assumption that $\mathsf{root}^{\top}_{\top}$ in γ_1. *Property 5*: If a process p receives token_3, then by definition of token_3, $p^{?}_{\top}$ and $\mathsf{parent}(p)^{?}_{\bot}$. Following the argument of the satisfaction of Property 1, if $\mathsf{parent}(p)^{?}_{\bot}$ in γ_1, then $\mathsf{parent}(p)$ can be only in state $\mathsf{parent}(p)^{\bot}_{\bot}$. By Property 1-b, if $\mathsf{parent}(p)^{\bot}_{\bot}$, then p is in state $p^{\bot}_{\bot}$, implying that token_3 does not hold for p, which implies Property 5. □

Lemma A.15. Given a legConfig $\gamma_0 \in C1$ of a topology, for any step (γ_0, γ_1), if $\mathsf{root}^{\top}_{\bot}$ in γ_1, then $\gamma_1 \in C2 \cup C4$.

Proof. Let p_0 be the root. By definition, p_0 is in state ${p_0}^{\top}_{\top}$ in γ_0. p_0 switches into ${p_0}^{\top}_{\bot}$ in γ_1 only if the command gc_1 is enabled in γ_0. Since $\gamma_0 \in C1$, then by Definition 5.9, Property 2, m_0 is equal to μ_0 in γ_0. By gc_1, if p_0 points to itself, then m_0 is the maximum among p_0 and its children, implying that Property 4 holds in γ_1 and $\gamma_1 \in C4$. Otherwise, if p_0 points to one of its children ch, then Property 4 also holds. The other properties of Definition 5.9 are also satisfied in γ_1, implying that $\gamma_1 \in C2$. □

Lemma A.16. Given a legConfig γ_0 in $C2$, for any step (γ_0, γ_1), if $\gamma_1 \models \mathsf{root}^{\top}_{\bot}$, then $\gamma_1 \in C2 \cup C3$.

Proof. The proof is similar to the proof of Lemma A.14. By Definition A.2, since $\gamma_0 \in C2$, then $\mathsf{root}^{\top}_{\bot}$ holds in γ_0. It is shown that each property in Definition 5.9 holds in γ_1.

Property 1: Since Property 1 holds in γ_0, then there exists no process p in state $p^{\top}_{\top}$ in γ_0. Now whenever a process p_i switches from ${p_i}^{\bot}_{?}$ to ${p_i}^{\top}_{?}$, its is due to token_3 (Commands $\mathsf{gc}_{6-8}, \mathsf{gc}_{11-12}$). By Property 1-a,b, if a process has a top token then, ${p_i}^{?}_{\top}$ and $\mathsf{parent}(p_i)^{?}_{\bot}$, implying that p_i switches the value of x into $\top$ in γ_1, and Property 1-a holds. Similarly, whenever a process p_j switches from ${p_j}^{\top}_{?}$ to ${p_j}^{\bot}_{?}$, then it is due to bottom token, implying that p_j does not change the value of x. This implies that Property 1-b holds in γ_1. *Property 2*: By Property 1, there is no process p in state $p^{\top}_{\top}$ in γ_0. If a process p switches into $p^{\top}_{\top}$ in γ_1, then $p^{?}_{\bot}$ and $\mathsf{parent}(p)^{?}_{\top}$ in γ_0, which does not hold by Property 1. Therefore, there is no process p with a state $p^{\top}_{\top}$ in γ_1. *Property 3*: By definition of Property 1, no process p may switch into $p^{\bot}_{\top}$. *Property 4*: By assumption, the root does not point to itself, since $\gamma_0 \in C2$. Therefore, by Property 4, in γ_0, there exists a path $\mathsf{root} = p_0, \ldots, p_s$, where for $0 \leq i < s$, $p_i.\ell = i + 1$, $p_s.\ell = s$, and $\mathsf{m}_s = \mu_s$ is the maximum among all processes in $\mathcal{T}$. In addition, since $\gamma_0 \in C2$, then ${p_s}^{\bot}_{\top}s$. By Property 1, token_1 and token_2 cannot be enabled in γ_0. This implies that either token_3 or token_4 may be enabled in γ_0. If token_3 holds for any process in the path, then one of

$\mathsf{gc}_6, \mathsf{gc}_7, \mathsf{gc}_{11}, \mathsf{gc}_{12}$ is enabled, and the result of the command does not violate Property 4. In particular, if gc_6 or gc_{12} is enabled, then it is enabled only for p_s by definition, and the command switches p_s from $p_s{}^{\perp}_{\top}$ into $p_s{}^{\top}_{\perp}$ or $p_s{}^{\perp}_{\perp}$, implying that $\gamma_1 \in C3$. For any process p_j that is not in the path, if token_3 holds in γ_0, then by Property 4, the enabled commands may be either gc_8 or gc_{12}, and their result does not violate Property 4. Therefore, Property 4 holds in γ_1. *Property 5*: Following the argument of Property 4, the condition $\mathsf{token}_3 \wedge \mathsf{parent}(p_s).\ell = s \wedge p_s.\ell = s$ holds only for p_s, and either gc_6 or gc_{11} is enabled. The execution of any of gc_6, gc_{11} does not enable any of them in γ_1, implying that Property 5 holds in γ_1. □

Lemma A.17. Given a legConfig $\gamma_0 \in C2$, for each execution $\gamma_0, \gamma_1, \ldots$, there exists a configuration γ_e for $e > 0$, such that

- $\gamma_e \in C3$ and $\forall\, 0 \le i \le e-1 \bullet \gamma_i \in C2$, and
- for the step (γ_{e-1}, γ_e), there is exactly one process $p \neq \mathsf{root}$ such that $\gamma_{e-1} : p{}^{\perp}_{\top}p.\mathsf{id}$, $\gamma_e : p{}^{\top}_{\perp}p.\mathsf{id}$, and p runs the command *critSection*() in the step.

Proof. By Lemma A.16, since $\gamma_0 \in C2$, then there exists a path $p_0, \ldots, p_s$ such that:

$$p_0{}^{\top}_{\perp}1, p_1{}^{\top}_{\perp}2, \ldots, p_j{}^{\top}_{\perp}p_{j+1}, p_{j+1}{}^{\perp}_{\top}p_{j+2}, \ldots, p_s{}^{\perp}_{\top}p_s \tag{A.9}$$

By Algorithm 5.6 and Lemma 5.2: The processes $p_0, \ldots, p_j$, $p_{j+2}, \ldots, p_s$ have no top tokens. The processes $p_0, \ldots, p_s$ have no bottom tokens. The only process with a token is p_{j+1} (gc_7), and it is a top token. In γ_1, p_{j+1} switches into $p_{j+1}{}^{\top}_{\perp}j+2$. Note that $\gamma_1 \in C2$. Analogously, p_{j+2} performs the same action in γ_2. Since the number of $p_{j+1}, \ldots, p_s$ is finite then a configuration γ_{s-1} is reached where

$$p_0{}^{\top}_{\perp}1, p_1{}^{\top}_{\perp}2, \ldots, p_{s-1}{}^{\top}_{\perp}s, p_s{}^{\perp}_{\top}s \tag{A.10}$$

In γ_{e-1}, by Algorithm 5.6, the only process with a token is p_s (gc_6) which is a top token. In the execution step (γ_{e-1}, γ_e), p_s runs *critSection*() and switches into $p_s{}^{\top}_{\perp}s$, implying that $\gamma_e \in C3$ by definition of $C3$. □

Lemma A.18. Given a legConfig γ_0 in $C3$, for any step (γ_0, γ_1), if $\gamma_1 \models \mathsf{root}^{\top}_{\perp}$, then $\gamma_1 \in C3$.

Proof. Follows the proof argument of Lemma A.16. □

Lemma A.19. Given a legConfig $\gamma_0 \in C3$, for any execution step (γ_0, γ_1), if $\mathsf{root}^{\top}_{\top}$ in γ_1, then $\gamma_1 \in C1$.

Proof. By definition, the root is in state $\mathsf{root}^{\top}_{\perp}$ in γ_0. The root switches into $\mathsf{root}^{\top}_{\top}$ in γ_1 only if one of the commands gc_2 and gc_3 is enabled in γ_0. Since $\gamma_0 \in C3$, then the root does not point to itself, and gc_2 cannot be enabled. Therefore, gc_3 is enabled in γ_0, implying that each root's child ch is in state $ch^{\perp}_{\perp}$. By Definition 5.9, Property 1, each non-root process p is in state $p^{\perp}_{\perp}$. This implies that no process other than the root has a token in γ_0. When the root executes gc_3, the root sets m to $update_{\mathsf{m}}()$, satisfying Property 2 of Definition 5.9. The other properties are not violated since no process other than the root has a token. This implies that γ_1 is a legConfig, and since $\mathsf{root}^{\top}_{\top}$ in γ_1, then $\gamma_1 \in C1$. □

Lemma A.20. Given a legConfig γ_0 in *C4*, for any step (γ_0, γ_1), if $\gamma_1 \models \mathsf{root}_{\perp}^{\top}$, then $\gamma_1 \in$ *C4*.

Proof. Follows the proof argument of Lemma A.16. □

Lemma A.21. Given a legConfig $\gamma_0 \in$ *C4*, for any execution step (γ_0, γ_1), if $\mathsf{root}_{\top}^{\top}$ in γ_1, then $\gamma_1 \in$ *C1*.

Proof. Same as the proof argument of Lemma A.19, with a difference that the root executes the command gc_2 instead of gc_3. □

Proof of Theorem 5.6

Proof. By Lemmata A.14–A.21, for each configuration γ_i in any of the categories *C1*, *C2*, *C3*, *C4*, for each execution step (γ_i, γ_{i+1}), γ_{i+1} is also in one of the categories. By Lemma A.13, the union of the four categories is the set of all legConfigs. This implies the theorem. □

A.6 Proof of Theorem 5.7

Proof. Let d be $\mathsf{depth}(\mathcal{T})$. By Theorem 5.6, any configuration following γ_0 is a legConfig. By Lemma 5.3, the root switches into $\mathsf{root}_{\perp}^{\top}$ in $i \leq 2d$ steps. γ_i is either in *C2* or *C4*. From γ_i, by Lemmata A.17, A.19, and A.21, the root switches to $\mathsf{root}_{\top}^{\top}$ in $r \leq 2d$ steps, such that $\gamma_{i+r} \in$ *C1*. If $\gamma_i \in$ *C2*, then there exists $i < e < r$, such that $\gamma_e \in$ *C3*, and by Lemma A.17 exactly one non-root process is privileged in γ_{e-1}. Otherwise, $\gamma_i \in$ *C4* holds, and the root executes *critSection*() before reaching γ_r, by Lemma A.21. The sum of i and r is less than or equals $4d$. □

A.7 Proof of Lemma 5.4

Proof. The liveness property of *EUPS* holds by Theorem 5.7. It remains to show the safety property. First, unique process selection – *UPS* – is concerned. In Algorithm 5.6, the commands, whose actions include *critSection*(), are $\mathsf{gc}_2, \mathsf{gc}_6$ and gc_{11}. It is shown that in a legConfig, if a command is enabled for a process, then none of them is enabled for any other process:

1. If the command gc_2 is enabled, then by definition of gc_2, each root's child ch is in a state $ch_{\perp}^{\perp}$. By Definition 5.9, Property 1, each process p, in the maximal subtree rooted by ch, is in state $p_{\perp}^{\perp}$. This implies that token_3 is not enabled for p, implying that the commands $\mathsf{gc}_6, \mathsf{gc}_{11}$ are not enabled for p.

2. If any of gc_6 or gc_{11} is enabled for a process p_s, then (a) By Definition 5.9, Property 5, there is no other process with enabled gc_6 or gc_{11}. (b) By Definition 5.9, Property 1, there exists a root's child ch that is not in state $ch_{\perp}^{\perp}$, implying that gc_2 is not enabled.

The argument above shows that *UPS* holds. By Property 4 of Definition 5.9, it follows that if a process is privileged in a legConfig γ_i, then it has the maximum μ_i among all processes. This implies that *EUPS* holds in any legConfig. □

B Correctness Proofs for Chapter 6

B.1 Proof of Lemma 6.3

Proof. By induction over the depth of nodes. For the base case and the step, it is shown first "$\leq$" and then "$\geq$" to obtain (6.1).

Base case $d = 1$: Let v be a node with $\mathsf{depth}(v) = 1$. Let $\mathcal{I}$ be a synchronized evolution over $\mathcal{T}$ with slot length ω and least upper bound δ^{max} on the clock drift rates, and let $\mathcal{I}$ be scheduled by assignment assign. Let $t_2 \in \mathsf{Time}$ be a point in time. By Definition 6.7, there is a point $t_1 \leq t_2$ such that $\mathsf{clk}_v^{\mathcal{I}}(\cdot)$ is differentiable on the interval (t_1, t_2). By Lemma 6.2, $\varrho_v^{\mathcal{I}}(t_2) = \varrho_v^{\mathcal{I}}(t_1) + \int_{t_1}^{t_2} \delta_v^{\mathcal{I}}(t)\,dt$. Because $\mathsf{depth}(v) = 1$, the parent of v is the central unit and t_1 is a synchronization point, thus by Definitions 6.3 and 6.7, $\varrho_v^{\mathcal{I}}(t_1) = \mathsf{clk}_v^{\mathcal{I}}(t_1) - \mathsf{clk}_{\mathsf{cu}(\mathcal{T})}^{\mathcal{I}}(t_1) = 0$, and thus

$$\varrho_v^{\mathcal{I}}(t_2) = \int_{t_1}^{t_2} \delta_v^{\mathcal{I}}(t)\,dt. \tag{B.1}$$

Because δ^{max} is an upper bound on the clock drift rates in $\mathcal{I}$, it holds that $\varrho_v^{\mathcal{I}}(t_2) \leq (t_2 - t_1)\delta^{max}$, and thus by Lemma 6.1, $\varrho_v^{\mathcal{I}}(t_2) \leq (k+1) \cdot \omega\delta^{max}$. Given the set of synchronized evolutions which are scheduled by assign, have slot length ω, and for which δ^{max} is the least upper bound on the clock drift rates. Let v be a node where there is an evolution $\mathcal{I}$ and a point in time t with $\delta_v^{\mathcal{I}}(t) = \delta^{max}$, and with two synchronization points $t_1, t_2 \in \mathsf{Time}$ such that $t_2 - t_1 = (k+1) \cdot \omega$, and such that $\delta_v^{\mathcal{I}}$ is differentiable on (t_1, t_2). For this evolution, the equation (B.1) applies and yields $\varrho_v^{\mathcal{I}}(t_2) = (t_2 - t_1)\delta^{max} = (k+1) \cdot \omega \cdot \delta^{max}$, and thus $\varrho_v^{\mathcal{I}}(t_2) \geq (k+1) \cdot \omega\delta^{max}$.

Induction step $d' \to d' + 1$: Assume that the equation (6.1) holds for all nodes of depth up to d'. Let v be a node of depth $d' + 1$. Let $\mathcal{I}$ be a synchronized evolution over $\mathcal{T}$ with slot length ω and least upper bound δ^{max} on the clock drift rates, and let $\mathcal{I}$ be scheduled by assignment assign. Let $t_2 \in \mathsf{Time}$ be a point in time. By Definition 6.7, there is a synchronization point $t_1 \leq t_2$ such that $\mathsf{clk}_v^{\mathcal{I}}(\cdot)$ is differentiable on the interval (t_1, t_2). By Lemma 6.2, $\varrho_v^{\mathcal{I}}(t_2) = \varrho_{\mathsf{parent}(v)}^{\mathcal{I}}(t_1) + \int_{t_1}^{t_2} \delta_v^{\mathcal{I}}(t)\,dt$. The parent of v has depth d', thus by induction hypothesis,

$$\varrho_{\mathsf{parent}(v)}^{\mathcal{I}}(t_1) \leq \left(\sum_{i=1}^{d'-1} \mathsf{fdist}_{\mathsf{assign}}(v_i, v_{i+1}) + \mathsf{k} + 1 \right) \cdot \omega\delta^{max}. \tag{B.2}$$

Because δ^{max} is an upper bound on the clock drift rates in $\mathcal{I}$,

$$\int_{t_1}^{t_2} \delta_v^{\mathcal{I}}(t)\,dt \leq \mathsf{fdist}(\mathsf{parent}(v), v) \cdot \omega\delta^{max}, \tag{B.3}$$

thus, by the equations (B.2) and (B.3),

$$\begin{aligned} \varrho_v^{\mathcal{I}}(t_2) &\leq \left(\sum_{i=1}^{d'-1} \mathsf{fdist}_{\mathsf{assign}}(v_i, v_{i+1}) + \mathsf{k} + 1 \right) \cdot \omega\delta^{max} \\ &\quad + \mathsf{fdist}(\mathsf{parent}(v), v) \cdot \omega\delta^{max} \\ &= \left(\sum_{i=1}^{(d'+1)-1} \mathsf{fdist}_{\mathsf{assign}}(v_i, v_{i+1}) + \mathsf{k} + 1 \right) \cdot \omega\delta^{max}. \end{aligned} \tag{B.4}$$

Given the set of synchronized evolutions which are scheduled by assign, have slot length ω, and for which δ^{max} is the least upper bound on the clock drift rates, let v be a node where there is (by implicit induction hypothesis) an evolution $\mathcal{I}$ and a point in time t with

$$\delta_{\mathsf{parent}(v)}^{\mathcal{I}}(t) = \left(\sum_{i=1}^{d'-1} \mathsf{fdist}_{\mathsf{assign}}(v_i, v_{i+1}) + \mathsf{k} + 1 \right) \cdot \omega\delta^{max}. \tag{B.5}$$

Because parent(v) has depth d', and with $\delta_v^{\mathcal{I}}(t) = \delta^{max}$, and with two synchronization points $t_1, t_2 \in \mathsf{Time}$ such that $t_2 - t_1 = \mathsf{fdist}(\mathsf{parent}(v), v) \cdot \omega$. (also by implicit induction hypothesis: the node which satisfies the equation (B.5) is synchronized as late as possible, i.e. at the right end of its slot), for this evolution, the equation (B.1) applies and yields $\varrho_v^{\mathcal{I}}(t_2) = (t_2 - t_1)\delta^{max} = \mathsf{fdist}(\mathsf{parent}(v), v) \cdot \omega \cdot \delta^{max}$. Thus

$$\varrho_v^{\mathcal{I}}(t_2) \geq \left(\sum_{i=1}^{(d'+1)-1} \mathsf{fdist}_{\mathsf{assign}}(v_i, v_{i+1}) + \mathsf{k} + 1 \right) \cdot \omega \cdot \delta^{max}. \tag{B.6}$$

□

B.2 Proof of Lemma 6.4

PROOF. For topologies of depth 1, both claims hold trivially. Let $\mathcal{T}$ be a topology of depth $d \geq 2$.

Regarding Point (1): Let assign be an assignment. Given that each sensor is assigned to a slot by assign, it follows by Definition 6.10 that for each two different sensors $v, v' \in \mathsf{Sn}(\mathcal{T})$:

$$\mathsf{fdist}_{\mathsf{assign}}(v, v') \leq \mathsf{k} - 1, \tag{B.7}$$

thus for any path $v_0, \ldots, v_m$ in $\mathcal{T}$, where $m \leq \mathsf{depth}$,

$$\sum_{i=1}^{m-1} \mathsf{fdist}_{\mathsf{assign}}(v_i, v_{i+1}) \leq (d-1)(\mathsf{k}-1). \tag{B.8}$$

Regarding Point (2):
"$\longleftarrow$":
Let assign have a (6.2)-path (the path mentioned in the equation (6.2)). The following diagram illustrates the (6.2)-path for assign:

$$\ldots + v_d + v_{d-1} + \ldots + v_2 + v_1 + \ldots$$

that is, the children on the path are synchronized immediately before their parent. By the equation (6.2) and Definition 6.10,

$$\mathsf{fdist}_{\mathsf{assign}}(v_i, v_{i+1}) = \mathsf{k} - 1. \tag{B.9}$$

Thus

$$\sum_{i=1}^{d-1} \mathsf{fdist}_{\mathsf{assign}}(v_i, v_{i+1}) = (d-1)(\mathsf{k}-1) \tag{B.10}$$

for the (6.2)-path. Thus, with the equation (B.8),

$$\mathcal{D}_{\mathsf{assign}} = (d-1)(\mathsf{k}-1). \tag{B.11}$$

"$\longrightarrow$":
Let assign be an assignment with $\mathcal{D}_{\mathsf{assign}} = (d-1)(\mathsf{k}-1)$. by Definition 6.10 and by the equation (B.7), there is a path $v_0, v_1, \ldots, v_d$, $v_0 = \mathsf{cu}(\mathcal{T})$ where $\mathsf{fdist}_{\mathsf{assign}}(v_i, v_{i+1}) = (\mathsf{k}-1)$. Thus, by Definition 6.10, there is a (6.2)-path. □

B.3 Proof of Lemma 6.5

Proof. For topologies of depth 1, the two claims hold trivially. Let $\mathcal{T}$ be a topology of depth $d \geq 2$. First it is shown that if conditions (2a) and (2b) are satisfied, then $\mathcal{D}_{\mathsf{assign}} = s - 1$. Second, it is shown that if any of (2a) and (2b) is violated, then $\mathcal{D}_{\mathsf{assign}} > s - 1$.

Let assign be an assignment of slots to nodes such that the conditions (2a) and (2b) are satisfied. Let $v_1, \ldots, v_m$ with $(\mathsf{cu}(\mathcal{T}), v_1) \in \mathcal{E}$ be a path in $\mathcal{T}$. By the condition (2a), for each $1 < j \leq m$:

$$\mathsf{fdist}_{\mathsf{assign}}(v_1, v_j) \leq (s-1). \tag{B.12}$$

By the condition (2b), for each $1 \leq i < m - 1$:

$$\begin{aligned}\mathsf{fdist}_{\mathsf{assign}}(v_i, v_{i+2}) = \\ \mathsf{fdist}_{\mathsf{assign}}(v_i, v_{i+1}) + \mathsf{fdist}_{\mathsf{assign}}(v_{i+1}, v_{i+2}).\end{aligned} \tag{B.13}$$

By the equations (B.12) and (B.13):

$$\sum_{i=1}^{j-1} \mathsf{fdist}_{\mathsf{assign}}(v_i, v_{i+1}) = \mathsf{fdist}_{\mathsf{assign}}(v_1, v_j) \leq (s-1). \tag{B.14}$$

Let v_m be a sensor in a maximal subtree of $\mathcal{T}$ which is assigned the (modulo k) latest slot (this sensor must exist by Definition of subtree). Then

$$\mathsf{fdist}_{\mathsf{assign}}(v_1, v_m) = s - 1. \tag{B.15}$$

Thus, by the equations (B.14) and (B.15),

$$\mathcal{D}_{\mathsf{assign}} = s - 1. \tag{B.16}$$

It is shown that, if the conditions (2a) or (2b) are violated, then $\mathcal{D}_{\mathsf{assign}} > s-1$. (1) The condition (2a) is violated. Let assign' be an assignment such that the condition (2a) is violated. It follows that there exists a path $v_0, \ldots, v_m$, where $v_0 = \mathsf{cu}(\mathcal{T})$ and $m \leq d$, such that $\mathsf{fdist}_{\mathsf{assign}'}(v_1, v_m) > (s-1)$. Thus by Definition 6.10, $D_{\mathsf{assign}'} > (s-1)$. (2) The condition (2b) is violated. Let assign'' be an assignment such that the condition (2b) is violated. It follows that there exist three nodes $v, v', v'' \in \mathcal{V}$ such that $(v, v'), (v', v'') \in \mathcal{E}$ and $\mathsf{fdist}_{\mathsf{assign}''}(v, v') \geq \mathsf{fdist}_{\mathsf{assign}''}(v, v'')$. By Definition 6.5, assign is bijective, and therefore $\mathsf{fdist}_{\mathsf{assign}''}(v, v') \neq \mathsf{fdist}_{\mathsf{assign}''}(v, v'')$ holds. By Definition 6.10 and the order of v, v', v'', it follows that $\mathsf{fdist}_{\mathsf{assign}''}(v, v') + \mathsf{fdist}_{\mathsf{assign}''}(v', v'') > \mathsf{k}$. As v, v', v'' lie on a path in $\mathcal{T}$, $\mathcal{D}_{\mathsf{assign}} > \mathsf{k} > (s-1)$. □

B.4 Proof of Theorem 6.1

PROOF. The case where $\mathcal{T}$ has only one sensor is trivial. Thus in the following it is assumed that $\mathcal{T}$ has at least two sensors. Let $\mathcal{I}$ be a scheduled evolution over $\mathcal{T}$ with guard time ϕ. Each of the two points is considered separately.

Point (1): Let $v \in \mathsf{Sn}(\mathcal{T})$ be a sensor and let $t \in \mathsf{Time}$ be a point in time where v sends, i.e. where $\mathsf{send}_v^{\mathcal{I}}(t) = 1$. Because $\mathcal{I}$ is scheduled and has guard time ϕ, there is a slot $[t_1, t_2)$ of v and points in time $t_1', t_2' \in \mathsf{Time}$ such that $\mathsf{clk}_v^{\mathcal{I}}(t_1') = t_1 + \phi$, $\mathsf{clk}_v^{\mathcal{I}}(t_2') = t_2 - \phi$, and $t_1' \leq t < t_2'$ because by Definition 6.11, v sends a message only if the clock of v denotes a point within the σ-interval, i.e., between $t_1 + \phi, t_2 - \phi$. By Definition 6.3, it follows that $\mathsf{clk}_v^{\mathcal{I}}(t_1') = \mathsf{clk}_{\mathsf{cu}(\mathcal{T})}^{\mathcal{I}}(t_1') + \varrho_v^{\mathcal{I}}(t_1')$, and thus: $t_1 + \phi = \mathsf{clk}_{\mathsf{cu}(\mathcal{T})}^{\mathcal{I}}(t_1') + \varrho_v^{\mathcal{I}}(t_1')$. Because $\mathsf{clk}_{\mathsf{cu}(\mathcal{T})}^{\mathcal{I}}(t_1') = t_1'$ and using the premise, it follows that $t_1 \leq t_1 + \phi - \varrho_v^{\mathcal{I}}(t_1') = t_1'$. Analogously, $t_2' \leq t_2$, thus $t \in [t_1, t_2)$, which is the slot assigned to v by assign, and thus

$$\forall\, v \in \mathcal{V}, t \in \mathsf{Time} \bullet \mathsf{send}_v^{\mathcal{I}}(t) \longrightarrow \mathsf{slot}(t) = \mathsf{assign}(v). \tag{B.17}$$

It is shown that $\mathcal{I}$ does not have a message collision by contradiction. Assume that $\mathcal{I}$ has a message collision, i.e. there is a point in time $t \in \mathsf{Time}$ where two different nodes $v_1, v_2 \in \mathcal{V}$ send, i.e. where $\mathsf{send}_{v_1}^{\mathcal{I}}(t) \wedge \mathsf{send}_{v_2}^{\mathcal{I}}(t)$. By the equation (B.17), $\mathsf{slot}(t) = \mathsf{assign}(v_1) \wedge \mathsf{slot}(t) = \mathsf{assign}(v_2)$, and thus $\mathsf{assign}(v_1) = \mathsf{assign}(v_2)$, which is a contradiction to the fact that assign has one slot assigned to each sensor per frame.

Point (2): Let $v \in \mathsf{Sn}(\mathcal{T})$ be a sensor with parent v' and let $t \in \mathsf{Time}$ be a point in time where v sends, i.e. where $\mathsf{send}_v^{\mathcal{I}}(t) = 1$. By applying the same reasoning as in the proof of point (1) above, it follows that

$$t \in \left[t_1 + \tfrac{\phi}{2}, t_2 - \tfrac{\phi}{2}\right) \tag{B.18}$$

where $[t_1, t_2)$ is an assigned slot to v. The parent of v is listening throughout the slot of v. Because $\mathcal{I}$ is scheduled, there are points in time $t_1', t_2' \in \mathsf{Time}$ such that $\mathsf{clk}_{v'}^{\mathcal{I}}(t_1') = t_1$, $\mathsf{clk}_{v'}^{\mathcal{I}}(t_2') = t_2$, and

$$\forall\, t' \in [t_1', t_2') \bullet \mathsf{listen}_{v'}^{\mathcal{I}}(t'). \tag{B.19}$$

By Definition 6.3, $\mathsf{clk}_{v'}^{\mathcal{I}}(t_1') = \mathsf{clk}_{\mathsf{cu}(\mathcal{T})}^{\mathcal{I}}(t_1') + \varrho_{v'}^{\mathcal{I}}(t_1')$ holds, and thus $t_1 = \mathsf{clk}_{\mathsf{cu}(\mathcal{T})}^{\mathcal{I}}(t_1') + \varrho_{v'}^{\mathcal{I}}(t_1')$ holds. Because $\mathsf{clk}_{\mathsf{cu}(\mathcal{T})}^{\mathcal{I}}(t_1') = t_1'$ and using the premise it follows that $t_1' = t_1 - \varrho_v^{\mathcal{I}}(t_1') \leq t_1 + \frac{\phi}{2}$. Analogously it follows that $t_2' \geq t_2 - \frac{\phi}{2}$. Thus, using the equation (B.19),

$\forall t' \in \left[t_1 + \frac{\phi}{2}, t_2 - \frac{\phi}{2}\right) \bullet \mathsf{listen}^{\mathcal{I}}_{v'}(t')$ holds. Thus, using the equation (B.18), $\mathsf{listen}^{\mathcal{I}}_{v'}(t)$ holds. Thus there is no message loss at t. □

B.5 Proof of Lemma 6.6

Proof. Let $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ be a topology with at least two sensors.

Point (1): Let $\mathcal{I}$ be a scheduled evolution $\mathcal{I}$ over $\mathcal{T}$ with slot length ω. Let $0 < \tau < \omega$, v_1, v_2 be two nodes such that $\mathsf{assign}(v_2) = \mathsf{assign}(v_1) + 1$, $t \in \mathsf{Time}$ such that

$$t = t_2. \tag{B.20}$$

$[t_1, t_2)$ is the slot of v_1,

$$\varrho^{\mathcal{I}}_{v_1}(t) = -(\phi + \tau) \wedge \varrho^{\mathcal{I}}_{v_2}(t) = \phi, \tag{B.21}$$

and both nodes send continuously during their σ-interval. By Definition 6.3, $\varrho^{\mathcal{I}}_{v_1}(t) = \mathsf{clk}^{\mathcal{I}}_{v_1}(t) - \mathsf{clk}^{\mathcal{I}}_{\mathsf{cu}(\mathcal{T})}(t)$. Thus, using the premise (B.21) and (B.20), $\mathsf{clk}^{\mathcal{I}}_{v_1}(t) = \varrho^{\mathcal{I}}_{v_1}(t) + \mathsf{clk}^{\mathcal{I}}_{\mathsf{cu}(\mathcal{T})}(t) = t_2 - (\phi + \tau)$, which is in the σ-interval of v_2. Similarly, it follows that $\mathsf{clk}^{\mathcal{I}}_{v_2}(t) = \varrho^{\mathcal{I}}_{v_2}(t) + \mathsf{clk}^{\mathcal{I}}_{\mathsf{cu}(\mathcal{T})}(t) = t_2 + \phi$, which is in the σ-interval of v_2. Thus, using the premise that v_1 and v_2 send continuously within their σ-interval, it follows that $\mathsf{send}^{\mathcal{I}}_{v_1}(t) \wedge \mathsf{send}^{\mathcal{I}}_{v_2}(t)$ holds, i.e. there exists message collision between v_1 and v_2 at t.

Point (2): Let $\mathcal{I}$ be a scheduled evolution over $\mathcal{T}$ with slot length ω. Let $0 < \tau < \omega$, v be a node, $t \in \mathsf{Time}$ such that

$$t = t_2 - \frac{\phi}{2}. \tag{B.22}$$

$[t_1, t_2)$ is the slot of v,

$$\varrho^{\mathcal{I}}_{v}(t) = -\left(\frac{\phi}{2} + \tau\right) \wedge \varrho^{\mathcal{I}}_{\mathsf{parent}(v)}(t) = \frac{\phi}{2}, \tag{B.23}$$

and v sends continuously during its σ-interval. By Definition 6.3, $\varrho^{\mathcal{I}}_{v}(t) = \mathsf{clk}^{\mathcal{I}}_{v}(t) - \mathsf{clk}^{\mathcal{I}}_{\mathsf{cu}(\mathcal{T})}(t)$. Thus, using the premises (B.22) and (B.23), it follows that $\mathsf{clk}^{\mathcal{I}}_{v}(t) = \varrho^{\mathcal{I}}_{v}(t) + \mathsf{clk}^{\mathcal{I}}_{\mathsf{cu}(\mathcal{T})}(t) = t_2 - \frac{\phi}{2} - \left(\frac{\phi}{2} + \tau\right) = t_2 - \phi - \tau$, which is in the σ-interval of v. Thus $\mathsf{send}^{\mathcal{I}}_{v}(t) = 1$. holds. Similarly, it follows that $\mathsf{clk}^{\mathcal{I}}_{\mathsf{parent}(v)}(t) = \varrho^{\mathcal{I}}_{\mathsf{parent}(v)}(t) + \mathsf{clk}^{\mathcal{I}}_{\mathsf{cu}(\mathcal{T})}(t) = t_2 - \frac{\phi}{2} + \frac{\phi}{2} = t_2$, which is not in the slot assigned to v, thus $\mathsf{listen}^{\mathcal{I}}_{\mathsf{parent}(v)}(t) = 0$ holds, i.e. there exists message loss at t. □

B.6 Proof of Theorem 6.2

Proof. Looking for a safe guard time for $\mathcal{T}$ wrt. δ^{max}, implies – by Definition 6.12 – looking for $\phi \in \mathbb{R}^+_0$ such that each synchronized evolution $\mathcal{I}$ over $\mathcal{T}$ which is scheduled by an assignment assign and which has maximum drift rate δ^{max} and guard time ϕ exhibits neither message collision nor message loss.

Let $\mathcal{T}$ be a topology of depth d with $\mathsf{k} \in \mathbb{N}$ sensors, and upper bound $\delta^{max} \in \mathbb{R}^+_0$ on clock drift rates. Let $\sigma \in \mathbb{R}^+$ be the length of the σ-intervals, and assign be an assignment of slots to nodes. By Corollary 6.2, $\varrho^{max}_{\omega,\delta^{max}}(\mathsf{assign}) \leq (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1) \cdot \omega \cdot \delta^{max}$ holds. By

Corollary 6.3 and Lemma 6.6, a sufficient and necessary criterion for $\phi \in \mathsf{Time}$ being a safe guard time is that for each evolution $\mathcal{I}$, $\forall\, v \in \mathcal{V}, t \in \mathsf{Time} \bullet |\varrho_v^{\mathcal{I}}(t)| \leq \frac{\phi}{2}$ holds. By Definition 6.11, $\omega = 2\phi + \sigma$, thus ϕ is safe if $(\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1) \cdot (2\phi + \sigma) \cdot \delta^{max} \leq \frac{\phi}{2}$ holds. Simplification (and reordering) yields

$$
\begin{aligned}
& \delta^{max} \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1) \cdot (2\phi + \sigma) \cdot \leq \frac{\phi}{2} \\
\iff & 2 \cdot \delta^{max} \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1) \cdot (2\phi + \sigma) \leq \phi \\
\iff & 2 \cdot \delta^{max} \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1) \cdot 2\phi \\
& + 2 \cdot \delta^{max} \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1) \cdot \sigma \leq \phi \\
\iff & 4\phi \cdot \delta^{max} \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1) \\
& + 2 \cdot \delta^{max} \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1) \cdot \sigma \leq \phi \\
\iff & 2 \cdot \delta^{max} \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1) \cdot \sigma \\
& \leq (1 - 4 \cdot \delta^{max} \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1)) \cdot \phi.
\end{aligned} \tag{B.24}
$$

Distinguish two cases: Case $\delta^{max} = 0$: the equation (B.24) has the unique solution $\phi = 0$. Case: $\delta^{max} > 0$: there is a solution only if $(1 - 4 \cdot \delta^{max} \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1)) \neq 0$ holds, otherwise the intended division is not defined. Distinguish two cases:

- For $(1 - 4 \cdot \delta^{max} \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1)) > 0$, from the equation (B.24) it follows that

  $$\frac{2 \cdot \delta^{max} \cdot (\mathcal{D}_{\mathsf{assign}} + k + 1) \cdot \sigma}{1 - 4 \cdot \delta^{max} \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1)} \leq \phi$$

  by assumption, and by having $\mathcal{D}_{\mathsf{assign}} > 0$, $\delta^{max} > 0$, $d > 0$, $\mathsf{k} > 0$. The optimal, smallest choice of ϕ which satisfies this inequation would be just

  $$\sigma \cdot \frac{2 \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1) \cdot \delta^{max}}{1 - 4 \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1) \cdot \delta^{max}} \tag{B.25}$$

  which is in particular greater or equal to 0, thus a proper guard time. The claimed bound on δ^{max} is obtained from $(1 - 4 \cdot \delta^{max} \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1)) > 0$.

- For $(1 - 4 \cdot \delta^{max} \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1)) < 0$, it follows from the equation (B.24)

  $$\phi \leq \frac{2 \cdot \delta^{max} \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1) \cdot \sigma}{1 - 4 \cdot \delta^{max} \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1)} \tag{B.26}$$

  (because both sides are divided by a negative number). It follows that $2 \cdot \delta^{max} \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1) \cdot \sigma > 0$, because $\delta^{max} > 0$, $d > 0$, $\mathsf{k} > 0$, $\sigma > 0$, and thus

  $$\frac{2 \cdot \delta^{max} \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1) \cdot \sigma}{1 - 4 \cdot \delta^{max} \cdot (\mathcal{D}_{\mathsf{assign}} + \mathsf{k} + 1)} < 0. \tag{B.27}$$

  In this case there is no proper (positive) guard time. □

List of Figures

2.1 Examples of graphs of size $\mathsf{n} = 6$ . . . 11
2.2 Examples of trees of size $\mathsf{n} = 15$ and depth 3 . . . 12
2.3 An evolution $\mathcal{I}$ . . . 16

3.1 Convergence wrt. many properties . . . 18
3.2 con_Δ-Convergence . . . 21
3.3 con_Δ-WarmUp . . . 22

4.1 Finite incrementing system . . . 32
4.2 Execution of Algorithm 4.1 over a topology . . . 34

5.1 Desired scenario . . . 53
5.2 Scenario of Algorithm 5.5 . . . 58
5.3 Scenario of Algorithm 5.6 . . . 61
5.4 Snapshot correction. . . . 63
5.5 Categories of legitimate configurations for Algorithm 5.5 . . . 68
5.6 Categories of legitimate configurations for Algorithm 5.6 . . . 71
5.7 Performance of Algorithm 5.5 vs. the virtual ring approach . . . 74

6.1 TDMA frames and slots . . . 82
6.2 Scheduled evolution for sending (slot assignment) . . . 83
6.3 Example of worst and best assignments . . . 87
6.4 Guard time . . . 90
6.5 Safe guard time . . . 92
6.6 Wireless fire alarm system . . . 98
6.7 Time slot of the WFAS . . . 99
6.8 Verification of derived guard times . . . 104
6.9 Uppaal models – timed automata . . . 105

7.1 Topologies, over which Algorithm 4.1 is tested . . . 118

8.1 Wrap-up . . . 122

List of Tables

3.1 Executions converging to $\mathsf{con}_{0.25}$ 21

4.1 Time and space complexity for Algorithms 4.1–4.3 45

5.1 Unique process selection of some sorts of algorithms 73
5.2 Educated unique process selection of some sorts of algorithms 73

6.1 Optimal safe guard times and their effect on time and energy consumption 104

7.1 Model checking recurrence properties for Algorithm 4.2 119

List of Algorithms

4.1 Mutual exclusion algorithm based on the classical finite incrementing system 34
4.2 Mutual exclusion algorithm with optimal ME-convergence time $\lceil \mathsf{diam}(\mathcal{G})/2 \rceil - 1$ 35
4.3 Mutual exclusion algorithm with optimal ME-convergence time, and achieving 1.0 recurrence of *privileged* 36

5.4 Extended 4-state-machine algorithm to trees 54
5.5 Unique process selection by exploiting synchronicity for immediate feedback in PIF 57
5.6 Self-stabilizing PIF for educated unique process selection 60
5.7 Extension of Algorithm 5.6 60

Bibliography

[Abr85] Norman M. Abramson. Development of the ALOHANET. *IEEE Transactions on Information Theory*, 31(2):119–123, 1985.

[AD94] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AK05] Mahesh Arumugam and Sandeep S. Kulkarni. Self-stabilizing Deterministic TDMA for Sensor Networks. In *Proceedings of the Second International Conference on Distributed Computing and Internet Technology (ICDCIT)*, volume 3816 of *LNCS*, pages 69–81. Springer, 2005.

[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.

[AS87] Bowen Alpern and Fred B. Schneider. Recognizing Safety and Liveness. *Distributed Computing*, 2(3):117–126, 1987.

[Awe87] Baruch Awerbuch. Optimal Distributed Algorithms for Minimum Weight Spanning Tree, Counting, Leader Election and Related Problems (Detailed Summary). In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 230–240. ACM, 1987.

[BB11] Joffroy Beauquier and Janna Burman. Self-stabilizing Mutual Exclusion and Group Mutual Exclusion for Population Protocols with Covering. In *Proceedings of the 15th International Conference on Principles of Distributed Systems (OPODIS)*, volume 7109 of *LNCS*, pages 235–250. Springer, 2011.

[BDGM02] Joffroy Beauquier, Ajoy K. Datta, Maria Gradinariu, and Frédéric Magniette. Self-Stabilizing Local Mutual Exclusion and Daemon Refinement. *Chicago Journal of Theoretical Computer Science*, 2002, 2002.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on Uppaal. In *Revised Lectures of the Second International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Formal Methods for the Design of Real-Time Systems (SFM-RT)*, volume 3185 of *LNCS*, pages 200–236. Springer, 2004.

[BDPV99a] Alain Bui, Ajoy K. Datta, Franck Petit, and Vincent Villain. Optimal PIF in Tree Networks. In *Records of the 2nd International Meeting on Distributed*

Data & Structures (WDAS), volume 6 of *Proceedings in Informatics*, pages 1–16. Carleton Scientific, 1999.

[BDPV99b] Alain Bui, Ajoy K. Datta, Franck Petit, and Vincent Villain. Snap-Stabilizing PIF Algorithm in Trees. In *Proceedings of the 6th International Colloquium on Structural Information & Communication Complexity (SIROCCO)*, pages 32–46. Carleton Scientific, 1999.

[BDPV99c] Alain Bui, Ajoy K. Datta, Franck Petit, and Vincent Villain. Space Optimal PIF Algorithm: Self-Stabilized with no Extra Space. In *Proceedings of the 18th IEEE International Performance Computing and Communications Conference (IPCCC)*, pages 20–26. IEEE, 1999.

[BDPV99d] Alain Bui, Ajoy K. Datta, Franck Petit, and Vincent Villain. State-Optimal Snap-Stabilizing PIF in Tree Networks. In *Proceedings of the 1999 ICDCS Workshop on Self-stabilizing Systems (WSS)*, pages 78–85. IEEE Computer Society, 1999.

[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.

[BGW89] Geoffrey M. Brown, Mohamed G. Gouda, and Chuan-lin Wu. Token Systems that Self-Stabilize. *IEEE Transactions on Computers*, 38(6):845–852, 1989.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*, volume 26202649. MIT press Cambridge, 2008. ISBN: 978-0262026499.

[BPV04] Christian Boulinier, Franck Petit, and Vincent Villain. When Graph Theory Helps Self-Stabilization. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 150–159. ACM, 2004.

[BPV08] Christian Boulinier, Franck Petit, and Vincent Villain. Synchronous vs. Asynchronous Unison. *Algorithmica*, 51(1), 2008.

[BRT14] Jan Steffen Becker, Dilshod Rahmatov, and Oliver Theel. Region-Adherent Algorithms: Restricting the Impact of Faults on Service Quality. In *Proceedings of the 20th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 203–212. IEEE, 2014.

[CBRZ01] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[CCD+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv Symbolic Model Checker. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*, volume 8559 of *LNCS*, pages 334–342. Springer, 2014.

[CCGR99] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A New Symbolic Model Verifier. In *Proceedings of the 11th International Conference Aided Verification (CAV)*, volume 1633 of *LNCS*, pages 495–499. Springer, 1999.

[CD94] Zeev Collin and Shlomi Dolev. Self-Stabilizing Depth-First Search. *Information Processing Letters*, 49(6):297–301, 1994.

[CDDL15] Fabienne Carrier, Ajoy K. 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)*, pages 3:1–3:10. ACM, 2015.

[CDPV02] Alain Cournier, Ajoy K. Datta, Franck Petit, and Vincent Villain. Snap-Stabilizing PIF Algorithm in Arbitrary Networks. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, pages 199–206. IEEE Computer Society, 2002.

[CDPV05] Alain Cournier, Ajoy K. Datta, Franck Petit, and Vincent Villain. Optimal Snap-Stabilizing PIF Algorithms in Un-Oriented Trees. *Journal of High Speed Networks*, 14(2):185–200, 2005.

[CDV06] Alain Cournier, Stéphane Devismes, and Vincent Villain. Snap-Stabilizing PIF and Useless Computations. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 39–48. IEEE Computer Society, 2006.

[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)*, pages 486–493. IEEE Computer Society, 1992.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT press, 1999. ISBN: 978-0262032704.

[Cha82] Ernest J. H. Chang. Echo Algorithms: Depth Parallel Operations on General Graphs. *IEEE Transactions on Software Engineering (TSE)*, 8(4):391–401, 1982.

[CP00] Sébastien Cantarell and Franck Petit. Self-Stabilizing Group Mutual Exclusion for Asynchronous Rings. In *Proceedings of the 4th International Conference on Principles of Distributed Systems (OPODIS)*, Studia Informatica Universalis, pages 71–90. Suger, Saint-Denis, rue Catulienne, France, 2000.

[CP09] Clayton W. Commander and Panos M. Pardalos. A Combinatorial Algorithm for the TDMA Message Scheduling Problem. *Computational Optimization and Applications*, 43(3):449–463, 2009.

[CPVD01] Alain Cournier, Franck Petit, Vincent Villain, and Ajoy K. Datta. Self-Stabilizing PIF Algorithm in Arbitrary Rooted Networks. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS)*, pages 91–98. IEEE Computer Society, 2001.

[CSZ10] Viacheslav Chernoy, Mordechai Shalom, and Shmuel Zaks. On the Performance of Dijkstra's Third Self-Stabilizing Algorithm for Mutual Exclusion and Related Algorithms. *Distributed Computing*, 23(1):43–60, 2010.

[CYH91] Nian-Shing Chen, Hwey-Pyng Yu, and Shing-Tsaan Huang. A Self-Stabilizing Algorithm for Constructing Spanning Trees. *Information Processing Letters*, 39(3):147–151, 1991.

[DDHL09] Ajoy K. Datta, Stéphane Devismes, Florian Horn, and Lawrence L. Larmore. Self-Stabilizing k-out-of-ℓ Exclusion on Tree Networks. In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8. IEEE, 2009.

[DG13] Swan Dubois and Rachid Guerraoui. Introducing Speculation in Self-Stabilization – An Application to Mutual Exclusion. In *Proceedings of the 32nd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 290–298. ACM, 2013.

[DGS96] Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory Requirements for Silent Stabilization (Extended Abstract). In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 27–34. ACM, 1996.

[DGT00] Ajoy K. Datta, Maria Gradinariu, and Sébastien Tixeuil. Self-Stabilizing Mutual Exclusion Using Unfair Distributed Scheduler. In *Proceedings of the 14th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 465–470. IEEE Computer Society, 2000.

[DH97] Shlomi Dolev and Ted Herman. Superstabilizing Protocols for Dynamic Distributed Systems. *Chicago Journal of Theoretical Computer Science*, 1997, 1997.

[Dij65] Edsger W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, 8(9):569, 1965.

[Dij74] Edsger W. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Communications of the ACM*, 17(11):643–644, 1974.

[Dij86] Edsger W. Dijkstra. A Belated Proof of Self-Stabilization. *Distributed Computing*, 1(1):5–6, 1986.

[DIM91] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Uniform Dynamic Self-Stabilizing Leader Election (Extended Abstract). In *Proceedings of the 5th International Workshop on Distributed Algorithms (WDAG)*, volume 579 of *LNCS*, pages 167–180. Springer, 1991.

[DIM93] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity. *Distributed Computing*, 7(1):3–16, 1993.

[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.

[DIN05] DIN e.V. Fire Detection and Fire Alarm Systems – Part 25: Components Using Radio Links and System Requirements, German Version EN 54-25:2005, 2005.

[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.

[DKS10] Shlomi Dolev, Ronen I. Kat, and Elad Michael Schiller. When Consensus Meets Self-Stabilization. *Journal of Computer and System Sciences (JCSS)*, 76(8):884–900, 2010.

[DLL62] Martin Davis, George Logemann, and Donald W. Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5(7):394–397, 1962.

[dMB11] Leonardo M. de Moura and Nikolaj Bjørner. Satisfiability Modulo Theories: Introduction and Applications. *Communications of the ACM*, 54(9):69–77, 2011.

[Dol00] Shlomi Dolev. *Self-Stabilization.* The MIT Press, 2000. ISBN: 978-0262041782.

[DP60] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.

[DT10] Abhishek Dhama and Oliver Theel. A Transformational Approach for Designing Scheduler-Oblivious Self-stabilizing Algorithms. In *Proceedings of the 12th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6366 of *LNCS*, pages 80–95. Springer, 2010.

[DTW06] Abhishek Dhama, Oliver Theel, and Timo Warns. Reliability and Availability Analysis of Self-stabilizing Systems. In *Proceedings of the 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 4280 of *LNCS*, pages 244–261. Springer, 2006.

[EGE02] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-Grained Network Time Synchronization Using Reference Broadcasts. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI)*. USENIX Association, 2002.

[FB14] Fathiyeh Faghih and Borzoo Bonakdarpour. SMT-Based Synthesis of Distributed Self-stabilizing Systems. In *Proceedings of the 16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 8756 of *LNCS*, pages 165–179. Springer, 2014.

[FBT13] Narges Fallahi, Borzoo Bonakdarpour, and Sébastien Tixeuil. Rigorous Performance Evaluation of Self-Stabilization Using Probabilistic Model Checking. In *Proceedings of the 32nd International Symposium on Reliable Distributed Systems (SRDS)*, pages 153–162. IEEE Computer Society, 2013.

[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)*, pages 234–254. IEEE Computer Society, 1979.

[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.

[Gär99] Felix C. Gärtner. Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments. *ACM Computing Surveys*, 31(1):1–26, 1999.

[Gav72] Fanica Gavril. Algorithms for Minimum Coloring, Maximum Clique, Minimum Covering by Cliques, and Maximum Independent Set of a Chordal Graph. *SIAM Journal on Computing*, 1(2):180–187, 1972.

[GGHP96] Sukumar Ghosh, Arobinda Gupta, Ted Herman, and Sriram V. Pemmaraju. Fault-Containing Self-Stabilizing Algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 45–54. ACM, 1996.

[GH90] Mohamed G. Gouda and Ted Herman. Stabilizing Unison. *Information Processing Letters*, 35(4):171–175, 1990.

[GK93] Sukumar Ghosh and Mehmet H. Karaata. A Self-Stabilizing Algorithm for Coloring Planar Graphs. *Distributed Computing*, 7(1):55–59, 1993.

[HBTH14] Abdallah Hamini, Jean-Yves Baudais, Andrea M. Tonello, and Jean-François Hélard. Guard Time Optimization for Capacity Maximization of BPSK Impulse UWB Communications. *JCM*, 9(2):188–197, 2014.

[HMR94] Jean-Michel Hélary, Achour Mostéfaoui, and Michel Raynal. A General Scheme for Token- and Tree-Based Distributed Mutual Exclusion Algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 5(11):1185–1196, 1994.

[HP92] Debra Hoover and Joseph Poole. A Distributed Self-Stabilizing Solution to the Dining Philosophers Problem. *Information Processing Letters*, 41(4):209–213, 1992.

[HT04] Ted Herman and Sébastien Tixeuil. A Distributed TDMA Slot Assignment Algorithm for Wireless Sensor Networks. In *Proceedings of the First International Workshop on Algorithmic Aspects of Wireless Sensor Networks (ALGOSENSORS)*, volume 3121 of *LNCS*, pages 45–58. Springer, 2004.

[HZ06] Ted Herman and Chen Zhang. Best Paper: Stabilizing Clock Synchronization for Wireless Sensor Networks. In *Proceedings of the 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 4280 of *LNCS*, pages 335–349. Springer, 2006.

[Joh02] Colette Johnen. Service Time Optimal Self-Stabilizing Token Circulation Protocol on Anonymous Unidirectional Rings. In *Proceedings of the 21st International Symposium on Reliable Distributed Systems (SRDS)*, page 80. IEEE Computer Society, 2002.

[Joh04] Colette Johnen. Bounded Service Time and Memory Space Optimal Self-Stabilizing Token Circulation Protocol on Unidirectional Rings. In *Proceedings of the 18th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 2004.

[Jou98] Yuh-Jzer Joung. Asynchronous Group Mutual Exclusion (Extended Abstract). In *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 51–60. ACM, 1998.

[KA03a] Sandeep S. Kulkarni and Umamaheswaran Arumugam. Collision-Free Communication in Sensor Networks. In *Proceedings of the 6th International Symposium on Self-Stabilizing Systems (SSS)*, volume 2704 of *LNCS*, pages 17–31. Springer, 2003.

[KA03b] Sandeep S. Kulkarni and Umamaheswaran Arumugam. Transformations for Write-All-with-Collision Model. In *Proceedings of the 7th International Conference on Principles of Distributed Systems (OPODIS)*, volume 3144 of *LNCS*, pages 184–197. Springer, 2003.

[KA04] Sandeep S. Kulkarni and Umamaheswaran Arumugam. TDMA service for sensor networks. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS) Workshops*, pages 604–609. IEEE Computer Society, 2004.

[Kar72] Richard M. Karp. Reducibility Among Combinatorial Problems. In *Proceedings of a Symposium on the Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[KE14] Alex Klinkhamer and Ali Ebnenasir. Synthesizing Self-stabilization through Superposition and Backtracking. In *Proceedings of the 16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 8756 of *LNCS*, pages 252–267. Springer, 2014.

[KG93] Hermann Kopetz and Günter Grünsteidl. TTP – A Time-Triggered Protocol for Fault-Tolerant Real-Time Systems. In *Digest of Papers: The 23rd Annual International Symposium on Fault-Tolerant Computing (FTCS)*, pages 524–533. IEEE Computer Society, 1993.

[KK13] Alex Kravchik and Shay Kutten. Time Optimal Synchronous Self Stabilizing Spanning Tree. In *Proceedings of the 27th International Symposium on Distributed Computing (DISC)*, volume 8205 of *LNCS*, pages 91–105. Springer, 2013.

[KKM15] Amos Korman, Shay Kutten, and Toshimitsu Masuzawa. Fast and Compact Self-Stabilizing Verification, Computation, and Fault Detection of an MST. *Distributed Computing*, 28(4):253–295, 2015.

[KP90] Shmuel Katz and Kenneth J. Perry. Self-Stabilizing Extensions for Message-Passing Systems. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 91–101. ACM, 1990.

[KRS84] Ephraim Korach, Doron Rotem, and Nicola Santoro. Distributed Algorithms for Finding Centers and Medians in Networks. *ACM Transactions on Programming Languages and Systems*, 6(3):380–401, 1984.

[Kru79] H. S. M. Kruijer. Self-Stabilization (in Spite of Distributed Control) in Tree-Structured Systems. *Information Processing Letters*, 8(2):91–95, 1979.

[KT10] Sven Köhler and Volker Turau. A New Technique for Proving Self-stabilizing under the Distributed Scheduler. In *Proceedings of the 12th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6366 of *LNCS*, pages 65–79. Springer, 2010.

[KT12] Sven Köhler and Volker Turau. Fault-containing Self-stabilization in Asynchronous Systems with Constant Fault-Gap. *Distributed Computing*, 25(3):207–224, 2012.

[KY02] Hirotsugu Kakugawa and Masafumi Yamashita. Self-Stabilizing Local Mutual Exclusion on Networks in which Process Identifiers are not Distinct. In *Proceedings of the 21st International Symposium on Reliable Distributed Systems (SRDS)*, pages 202–211. IEEE Computer Society, 2002.

[Lam74] Leslie Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, 17(8):453–455, 1974.

[Lam86] Leslie Lamport. The Mutual Exclusion Problem: Part II – Statement and Solutions. *Journal of the ACM*, 33(2):327–348, 1986.

[LMV14] Florence Levé, Khaled Mohamed, and Vincent Villain. Snap-Stabilizing PIF on Non-oriented Trees and Message Passing Model. In *Proceedings of the 16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 8756 of *LNCS*, pages 299–313. Springer, 2014.

[Lya07] Aleksandr Lyapunov. Problème Général de la Stabilité du Mouvement. In *Annales de la Faculté des Sciences de Toulouse*, volume 9, pages 203–474, 1907. (A translation of the paper is published in the *Communications of the Kharkov Mathematical Society*, 1893, and reprinted in volume 17 of *Annals Mathematics Studies*, Princeton University Press, 1947).

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996. ISBN: 978-1558603486.

[Mil91] David L. Mills. Internet Time Synchronization: The Network Time Protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, 1991.

[MS90] Renato E. Mirollo and Steven H. Strogatz. Synchronization of Pulse-Coupled Biological Oscillators. *SIAM Journal on Applied Mathematics*, 50(6):1645–1662, 1990.

[NKM06] Yoshihiro Nakaminami, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. An Advanced Performance Analysis of Self-Stabilizing Protocols: Stabilization Time with Transient Faults During Convergence. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2006.

[OD08] Ernst-Rüdiger Olderog and Henning Dierks. *Real-Time Systems: Formal Specification and Automatic Verification*. Cambridge University Press, 2008. ISBN: 978-0521883337.

[ODT05] Jens Oehlerking, Abhishek Dhama, and Oliver Theel. Towards Automatic Convergence Verification of Self-stabilizing Algorithms. In *Proceedings of the 7th International Symposium on Self-Stabilizing Systems (SSS)*, volume 3764 of *LNCS*, pages 198–213. Springer, 2005.

[OR08] Waiel E. Osman and Tharek A. Rahman. Optimization of Guard Time Length for Mobile WiMAX System over Multipath Channel. In *Proceedings of the International MultiConference of Engineers and Computer Scientists*, volume 2. Citeseer, 2008.

[Par12] Dominique Paret. *FlexRay and its Applications: Real Time Multiplexed Network*. John Wiley & Sons, 2012. ISBN: 978-1119979562.

[Pet81] Gary L. Peterson. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, 12(3):115–116, 1981.

[PK00] Joel Phillips and Ken Kundert. An Introduction to Cyclostationary Noise. *The Designer's Guide Community*, 2000.

[PS13] Eldad Perahia and Robert Stacey. *Next Generation Wireless LANS: 802.11 n and 802.11 ac*. Cambridge university press, 2013. ISBN: 978-1107016767.

[PST14] Thomas Petig, Elad Schiller, and Philippas Tsigas. Self-Stabilizing TDMA Algorithms for Wireless Ad-hoc Networks without External Reference. In

Proceedings of the 13th Annual Mediterranean Ad Hoc Networking Workshop (MED-HOC-NET), pages 87–94. IEEE, 2014.

[PV00] Franck Petit and Vincent Villain. Optimality and Self-Stabilization in Rooted Tree Networks. *Parallel Processing Letters*, 10(1):3–14, 2000.

[PV07] Franck Petit and Vincent Villain. Optimal Snap-Stabilizing Depth-First Token Circulation in Tree Networks. *Journal of Parallel and Distributed Computing*, 67(1):1–12, 2007.

[Rap02] Theodore S. Rappaport. *Wireless Communications: Principles and Practice*, volume 2. Prentice Hall, 2002. ISBN: 978-0130422323.

[Ray86] Michel Raynal. *Algorithms for Mutual Exclusion.* The MIT Press, Cambridge, MA, 1986. ISBN: 978-0262181198.

[RH90] Michel Raynal and Jean-Michel Helary. *Synchronization and Control of Distributed Systems and Programs.* John Wiley & Sons Inc. 1990. ISBN: 978-0471924531.

[Seg83] Adrian Segall. Distributed Network Protocols. *IEEE Transactions on Information Theory*, 29(1):23–34, 1983.

[Str00] Ofer Strichman. Tuning SAT Checkers for Bounded Model Checking. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV)*, volume 1855 of *LNCS*, pages 480–494. Springer, 2000.

[Tel00] Gerard Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 2000. ISBN: 978-0521794831.

[The00a] Oliver Theel. A Verification Technique for Self-Stabilizing Algorithms Based on Ljapunov's "Second Method" (Brief Announcement). In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC)*, page 331. ACM, 2000.

[The00b] Oliver Theel. Exploitation of Ljapunov Theory for Verifying Self-Stabilizing Algorithms. In *Proceedings of the 14th International Symposium on Distributed Computing (DISC)*, volume 1914 of *LNCS*, pages 209–222. Springer, 2000.

[The01] Oliver Theel. A New Verification Technique for Self-Stabilizing Distributed Algorithms based on Variable Structure Systems and Ljapunov Theory. In *Proceedings of the 34th Hawaii International Conference on System Sciences (HICSS)*. IEEE Computer Society, 2001.

[Tix09] Sébastien Tixeuil. *Algorithms and Theory of Computation Handbook, chapter Self-stabilizing Algorithms.* Chapman & Hall/CRC Applied Algorithms and Data Structures. CRC Press, Taylor & Francis Group, 2009. ISBN: 978-1584888222.

[Tur14] Volker Turau. Analyzing the Fault-Containment Time of Self-Stabilizing Algorithms – A Case Study for Graph Coloring. *CoRR*, abs/1410.6669, 2014.

[TVS07] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms (Second Edition)*. Prentice-Hall, 2007. ISBN: 978-0132392273.

[WT06] Ernesto Wandeler and Lothar Thiele. Optimal TDMA Time Slot and Cycle Length Allocation for Hard Real-Time Systems. In *Proceedings of the 2006 Conference on Asia South Pacific Design Automation (ASP-DAC)*, pages 479–484. IEEE, 2006.

[WTP+05] Geoffrey Werner-Allen, Geetika Tewari, Ankit Patel, Matt Welsh, and Radhika Nagpal. Firefly-Inspired Sensor Network Synchronicity with Realistic Radio Effects. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 142–153. ACM, 2005.

Author's Publications

[1] Oday Jubran and Bernd Westphal. Formal Approach to Guard Time Optimization for TDMA. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems (RTNS)*, pages 223–233. ACM, 2013.

[2] Oday Jubran and Bernd Westphal. Optimizing Guard Time for TDMA in a Wireless Sensor Network – Case Study. In *Proceedings of the IEEE 39th Conference on Local Computer Networks (LCN) – Workshop Proceedings (9th SenseApp Workshop)*, pages 597–601. IEEE, 2014.

[3] Oday Jubran and Oliver Theel. Brief Announcement: Introducing Recurrence in Self-Stabilization. In *Proceedings of the 16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 8756 of *LNCS*, pages 352–354. Springer, 2014.

[4] Oday Jubran and Oliver Theel. Exploiting Synchronicity for Immediate Feedback in Self-Stabilizing PIF Algorithms. In *Proceedings of the 20th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 106–115. IEEE, 2014.

[5] Oday Jubran and Oliver Theel. A Self-stabilizing PIF Algorithm for Educated Unique Process Selection. In *Proceedings of the Third International Conference on Networked Systems (NETYS)*, volume 9466 of *LNCS*, pages 485–489. Springer, 2015.

[6] Oday Jubran, Eike Möhlmann, and Oliver Theel. Verifying Recurrence Properties in Self-stabilization by Checking the Absence of Finite Counterexamples. In *Proceedings of the 17th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 9212 of *LNCS*, pages 124–138. Springer, 2015.

[7] Oday Jubran and Oliver Theel. Recurrence in Self-Stabilization. In *Proceedings of the 34th International Symposium on Reliable Distributed Systems (SRDS)*, pages 58–67. IEEE, 2015.

Technical Reports

- Oday Jubran and Oliver Theel. Self-Stabilizing Mutual Exclusion Algorithm for Trees. Technical Report No. 91 of SFB/TR 14 AVACS, May 2013. `http://www.avacs.org/`.

- Oday Jubran and Oliver Theel. Introducing Recurrence in Self-Stabilization (Revised Version). Technical Report No. 101 of SFB/TR 14 AVACS, April 2015. `http://www.avacs.org/`.

Author's Biography

Oday Jubran, born on May 05, 1986, in Jerusalem

Education

Nov 2016	**Dr. rer. nat.** in Computer Science Carl von Ossietzky University of Oldenburg, Germany Thesis: *Recurrence in Self-Stabilization: Theory, Verification, and Application*
Jan 2012	**MSc.** in Computer Science Albert Ludwig University of Freiburg, Germany Thesis: *Formal Verification of a CNF Conversion in an SMT Solver*
May 2008	**BSc.** in Computer Information Systems Bethlehem University, Palestine Thesis: *A Decision Support System for Management and Marketing*
Jul 2004	**General Secondary Examination (Tawjihi)**, Scientific Branch Bethlehem, Palestine

Index

$active$, 55
α, 31
$aPath_\gamma$, 67
assign, 83
avg_depth, 50
Border, 40
c, 20
Ch, 11
$choose$, 59
clk, 81
coll, 81
con, 13
con_Δ, 20
$critSection()$, 55
cu($\mathcal{T}$), 80
$\mathcal{D}_{\mathsf{assign}}$, 87
degree, 10
δ, 82
δ^{max}, 85
Δ, 20
depth, 11
diam, 10
dist, 10
$\mathcal{E}$, 10
$ePath_\gamma$, 67
ϵ, 35
$EUPS$, 50
Evo($\mathcal{T}$), 81
$\mathsf{Evo}_{sync}(\mathcal{T}, \omega, \mathsf{assign}, \delta^{max})$, 85
fdist, 87
frm, 82
$\mathcal{G}$, 10
γ, 13
$\underline{\gamma}$, 20
Γ, 13
gc, 13
$\mathcal{I}$, 15
id, 10
idepth, 40
Inner, 11
$\mathcal{K}$, 31
ℓ, 55
l-$active$, 66
Leaves, 11
len (for path), 10
length (for execution), 13
listen, 81
loss, 81
m, 59
ME, 30
n, 10
$\mathcal{N}, \mathcal{N}^*$, 10
ω, 82
Ω, 13
$\mathcal{P}$, 14
parent, 11
$p_{\mathsf{x}}^{\mathsf{up}}\ell$, 56
ϕ, 90
φ, 31
$priv$, 42
$priv_\Delta$, 42
r, 31
Rec, 20
ϱ, 82
$\varrho^{max}_{\omega,\delta^{max}}$, 85
$safeSend$, 96
send, 81
σ-interval, 90
slot, 82, 96
$snapShot$, 62
Sn($\mathcal{T}$), 80
$\mathsf{stab}_\mathcal{X}$, 31
SU, 32
$\mathcal{T}$, 11

$\mathsf{tail}_{\mathcal{X}}$, 31
$\mathsf{tail}^*_{\mathcal{X}}$, 31
θ, 82
Time, 15
up, 53
$update_{\mathsf{m}}()$, 60
UPS, 49
$\mathcal{V}$, 10
w, 22
x, 53
$\mathcal{X}$, 31
Ξ, 13

active path, 67
assignment, 83
asynchronous execution, 14

border of an island, 40
bottom token, 64
branching factor, 11

central unit, 80
child, 11
clock drift, 82
clock speed, 82
command, 13
complete token, 52
configuration, 13
configuration space, 13
connected graph, 10
convergence of a real-time system, 15
convergence of an evolution, 15
convergence time wrt. con_Δ (discrete), 20
convergence time wrt. con, 15
convergence time wrt. con_Δ (real-time), 23
counterexample wrt. $(\mathsf{con}, \Delta, \mathsf{c})$, 112
critical section, 27

d-subtree, 11
degree of a vertex, 10
depth of a tree, 11
depth of a vertex, 11
depth of an island, 40
diameter of a graph, 10
directed graph, 10
distance between two vertices, 10
distributed scheduler, 14
distributed algorithm, 13
domain of a variable, 10
drift rate, 82

edge, 10
educated unique process selection, 50
empty snapshot, 62
enabled guarded command, 14
enabled process, 14
EN 54-25, 75
evolution, 15
evolution (TDMA), 81
execute token, 52
execution, 13
execution path, 67

fair execution, 14
fair scheduler, 14
feedback token, 52
finite execution, 13
finite incrementing system, 31
forward distance, 86

graph, 10
guarantee convergence time (discrete), 20
guarantee convergence time (real-time), 23
guarantee warmup time (real-time), 24
guarantee warmup time (discrete), 22
guard time, 90
guarded command, 13

have convergence time (discrete), 20
have warmup time (discrete), 22
highlighted snapshot, 62

inconsistent snapshot, 62
infinite execution, 13
init-island, 40
interpretation, 15
island, 39

label, 13
last active process, 66

legConfig, 67
legitimate configuration, 67
length of a path, 10
length of a slot, 82
length of an execution, 13
linked vertices, 10

maximal subtree, 11
maximum clock drift, 85
message collision, 81
message loss, 81
minimal counterexample, 112
mutex, 27
mutual exclusion, 30

neighbor, 10
node, 80
non-init-island, 40

parent, 11
path in a graph, 10
phase clock synchronizer, 29
PIF, 51
prefix of an execution, 13
process, 14

real-time system, 15
recurrence (discrete), 20
recurrence (real-time), 23
restrictions of an execution to a process, 44
root, 11

safe guard time, 91
scheduled evolution, 83
scheduler, 14
search token, 52
self-stabilizing real-time system, 15
self-stabilizing system (discrete), 14
sensor, 80
service time, 18
size of a graph, 10
state, 13
step of an execution, 13
strict subexecution, 13
sub-algorithm, 13
subexecution, 13
subtree, 11
suffix of an execution, 13
synchronization point, 84
synchronized evolution, 84
synchronous execution, 14
synchronous scheduler, 14
synchronous unison, 31
system (discrete), 13

TDMA schedule, 82
top token, 64
topology, 10
tree, 11

unique process selection, 49
unison, 29

variable of a vertex, 10
vertex, 10

warmup time (discrete), 22
warmup time (real-time), 24
wu-counterexample wrt. $(\mathsf{con}, \Delta, \mathsf{w})$, 112