Versatility and Efficiency in Self-Stabilizing Distributed Systems

Stéphane Devismes

HABILITATION À DIRIGER DES RECHERCHES

Spécialité : **Informatique**

Arrêté ministériel : 23 Novembre 1988

Présentée par

Stéphane Devismes

Habilitation préparée au sein du laboratoire **VERIMAG**
et de l'**École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Versatility and Efficiency in Self-Stabilizing Distributed Systems

Généralité et Efficacité dans les Systèmes Distribués Autostabilisants

Habilitation soutenue publiquement le **17 décembre 2020**,
devant le jury composé de :

Denis TRYSTRAM
Professeur, Grenoble INP, Président
Toshimitsu MASUZAWA
Professeur, Osaka University, Rapporteur
Achour MOSTÉFAOUI
Professeur, Université de Nantes, Rapporteur
Christian SCHEIDELER
Professeur, Universität Paderborn, Rapporteur
Carole DELPORTE-GALLET
Professeur, Université Denis Diderot Paris 7, Examinatrice
Jean-François MONIN
Professeur, Université Grenoble Alpes, Examinateur
Michel RAYNAL
Professeur émérite, Université de Rennes 1, Examinateur

« Il faut rougir de faire une faute, et non de la réparer.»[1]

Jean-Jacques Rousseau

[1] "You have to blush to make a mistake, not to fix it."

Disclaimer

This report is a feedback on my research activities over the last seventeen years. All the results presented here have been published in conference proceedings and journals. So, I presume (maybe wrongly) that they are technically sound.

Along these pages, I will try avoid to be too technical since in one hand my goal is rather to explain and popularize my results; and on the other hand I will try to put them into perspective. Hence, for every technical or formal detail I will simply link the reader back to my publications, these latter will be properly cited along this document.

Preamble

My research activities started in 2003, when I began my Master internship at Université de Picardie Jules Verne (Amiens, France) under the supervision of Alain Cournier and Vincent Villain. The main topic of this internship, and the PhD thesis that has followed, was *snap-stabilization* [Dev10], a fault-tolerance-related property applying to distributed systems. Since then, I have broadened my research topics first to self-stabilization, then fault tolerance in a wide meaning; and now my domain of interest encompasses every topic related to distributed algorithms, from the theoretical point of view (expressiveness, lower bounds, ...) to the practical applications (networking, security, ...). However, even if I have worked on failure detectors and robust fault tolerance [DGDF$^+$08, DDGFL10, DGDF10], robot and agent algorithms [CDPR11, DLP$^+$12, DPT13, DLPT19], homonymous systems [ADD$^+$16, ADD$^+$17a], random walks [ADGL12, ADGL14], security [ADJL17], and networking [FBA$^+$17], the heart of my research remains *self-stabilization and its variants*. Hence, this manuscript will be dedicated to this subject only. Actually, along the years, I have tried to conciliate two *a priori* orthogonal issues: *efficiency* and *versatility*. As commonly done in the algorithmic area, I aim at providing (self-) stabilizing solutions that are efficient in time and space complexities, but also in terms of fault tolerance (*e.g.*, snap-stabilizing solutions [BDP16]) or problem-related properties (*e.g.*, concurrency in resource allocation problems [CDDL15]). This topic not only consists in designing efficient algorithms, but also defining how to sharply evaluate efficiency, *e.g.*, by proposing new complexity measures [ADD17b] suited to the studied problem (or class of problems), or new stabilizing properties [AD17, ADDP19b] that more precisely capture the guarantees offered by a given algorithm. Versatility is also an important aspect of my researches. Classically, as an algorithmic designer, I try to propose algorithms requiring as less assumptions (*e.g.*, on the level of synchrony, level of anonymity, initial knowledge on the network topology, ...) as possible. Moreover, I not only consider classical distributed problems (*e.g.*, leader election [ACD$^+$17b], spanning tree construction [DKKT10]), but rather classes of problems by proposing parametric solutions [CDD$^+$15, DDH$^+$16] and other general schemes [DIJ19, CDD$^+$16]. These latter are especially interesting in terms of expressiveness since they are often used to characterize the class of problems admitting a specified solution [CDD$^+$16]. Finally, the term versatility also includes the search of lower bounds holding for wide classes of systems, *e.g.*, [BDGPBT09]. An important goal of this manuscript is to show that, in several interesting cases, efficiency and versatility can be unified in distributed self-stabilizing systems.

Contents

Chapter 1

Introduction

1.1 Distributed Systems

1.1.1 What is a Distributed System?

A *distributed system* is a set of interconnected autonomous computing entities that cooperate together to achieve a global aim [Tel01]. "Autonomous" means that each entity is endowed with its own private local control and memory. "Interconnected" means that each entity is able to exchange information with all or a part of other entities. The aim is global in the sense that it should require information transmissions between all or at least a large part of the entities. The key point is that, at each instant each entity only accesses a partial view of the global system configuration while transmissions and computations are typically asynchronous. In other words, in such a context, the challenge is that entities should compute and communicate to reach the global goal, despite their narrow local view and the maybe outdated received information.

The asynchronous assumption makes the programming of distributed systems harder but is motivated by several important considerations. First, it allows the designer to ignore specific timing characteristics that are maybe not available. Second, asynchronous algorithms are more general and so more portable than synchronous ones: they operate correctly whatever the timing guarantees. Finally, programming on the top of a synchronous unreliable system extensively makes use of timers that are maybe inaccurate if the time out occurs too quickly, or that drastically slow down the system if the period before the timer expiration is too large.

It is worth noticing that the previous definition of distributed systems does not only apply to computer networks, but also multiprocessor or multithreaded computers, and even some biological systems such as ant colonies.[1] However, it does not apply to some parallel architectures such as *SIMD* (Single Instruction Multiple Data, see Figure 1.1), which requires to perform the *same* operation on multiple data points simultaneously.

Here, we will focus on networks of computers only. Consequently, computing entities will be referred to as *processes* or *nodes*. The communication will be done either by message exchanges or using locally shared memories. The control part of each process will be described by one local algorithm, and the distributed algorithm will be the collection of all local algorithms.

[1] In this latter case, ants are the computing entities, the communication means are pheromones, sounds, and touch, and the global aim is basically the survival of the ant farm.

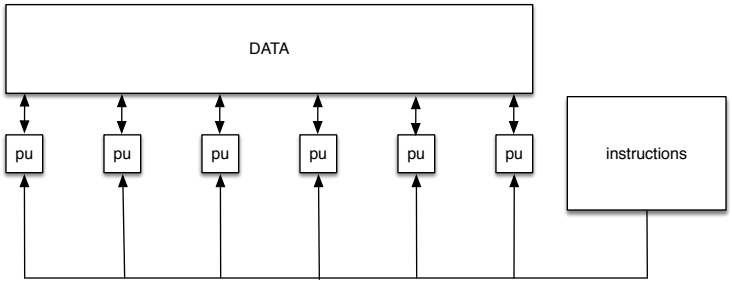


Figure 1.1: SIMD.

1.1.2 Distributed versus Non-distributed Systems

Consider first the difference between distributed and sequential (*a.k.a.*, central) systems. Maybe the two main differences between those two major kinds of system are the access to global knowledge and global time-frame. Indeed, in a sequential system, instruction executions are totally ordered (by their temporal occurrence) and can be based on the global configuration of the system. In contrast, in a distributed system, processes can access neither the whole configuration of the system, nor the global time. (Processes may hold a local clock, but any two local clocks might have different initial values and different drifts.) Now, as already stated, in distributed systems, process execution rate and communications are usually assumed to be asynchronous. Hence, there is no *a priori* synchronization mechanism and the events observable by a process can only be partially ordered (*cf.* the Lamport's causal order [Lam78]). This induces nondeterminism in the system execution: contrary to sequential deterministic algorithms, the execution of a distributed, yet deterministic, algorithm may be nondeterministic. For example, assume a network of three processes organized as a single line topology p_1, p_2, and p_3 and consider the simple deterministic protocol consisting for p_1 and p_3 to send their identifier, say 1 and 3, to p_2, and for p_2 to output the difference between the first and second received value. Since the system is asynchronous, two results are possible, -2 or 2, and a global observer have no mean to guess which of the two possible results will be output. This inherent nondeterminism in particular impacts the way problems are expressed. For example, in sequential deterministic algorithmics, a specification (*i.e.*, a formal definition of a problem) is usually defined as a (mathematical) function mapping inputs to outputs. In distributed computing, inputs and outputs are, with only a few exceptions, distributed. For example, while in a central system, the unique process can access the whole list of values in a sorting algorithm, in the corresponding distributed problem (usually called the *ranking* problem [DLDR13]), each value is given as input to exactly one process and the process does not compute the whole sorted list, but rather the rank of its input in the sorted list. Moreover, the problem to solve is usually defined in terms of *tasks* [MW87]. This latter is specified by an input/output relation, defining for each assignment of inputs to the system processes, the possible valid outputs of the processes. As an illustrative example, consider the (binary) *consensus* problem, which can be defined as follows. Each process is given a Boolean input value (0 or 1) and should *decide* a Boolean output fulfilling the following four requirements:

Agreement: If two processes decide, then they decide the same value.

Validity: Any decided value is the input value of some process.

Termination: Every process eventually decides.

Integrity: Every process decides at most once.

Then, the input/output relation specifies that only two outputs are possible: all-zero or all-one, and output all-zero (resp. all-one) is allowed only if at least one input is zero (resp. is one). In particular, when both zeros and ones are present among the inputs, two outputs are valid: all-zero or all-one, *i.e.*, for a given fixed assignment of inputs, a specification, like the one we give for the consensus, may not enforce a unique result.

Notice that contrary to parallelism where dividing the control into several computing entities is a choice motivated by an expected gain in performance, the fact that control is scattered in a distributed system is inherent to the nature of the architectures it models, *e.g.*, Internet, Cloud computing, Internet of Things, Wireless Sensor Networks, ...

To summarize, the distributed control, asynchrony, lack of global knowledge are important inherent parts of the numerous constraints in distributed systems, which make their design very challenging. Consequently, many researchers (myself included) are interested in the theoretical and algorithmic foundation of distributed computing which consists in investigating the various models, algorithms, and problems in terms of analytical complexity and expressive power (*i.e.*, power with respect to the set of problems that can be solved).

1.1.3 Distributed Problems

Rather than designing end-to-end distributed real-world applications, researchers focus on (very) basic building blocks that are commonly used in most of those applications. These basic problems can be classified into the following categories.

1.1.3.1 Data Exchange

This category in particular includes routing, broadcast, and propagation of information with feedback (PIF). It actually gathers all problems related to the transmission of a piece of information to all or part of the processes, even sometimes a single process.

1.1.3.2 Agreement

An agreement problem, *e.g.*, consensus, leader election, and k-set agreement, consists in making processes decide a local output in such way that all outputs satisfy a specific predicate.

1.1.3.3 Synchronization

Synchronization tasks aim at controlling the asynchronism of the system by making processes progress roughly at the same speed. This category includes unison, phase synchronization, and committee coordination.

1.1.3.4 Self-organization

The goal of a self-organizing algorithm is to build a structure on the top of the communication network (*e.g.*, a spanning tree or a clustering), or to organize processes (*e.g.*, group membership) to ease the communication exchanges.

1.1.3.5 Resource Allocation

Resource allocation consists in managing the concurrent access of processes to a typically small pool of non-shareable reusable resources, *e.g.*, printers, servers. Mutual exclusion, dining philosopher, and ℓ-exclusion are classical examples of resource allocation problems.

Notice that along the years, I have worked on all these aforementioned categories of problems. Precisely, concerning the data exchange problems, I have worked on PIF [CDV06b] and routing [ADJL17]. For the agreement problems, I have mainly worked on leader election [ACD+17b, ADD+16, ADD+17a], but also consensus [DGDF+08]. Unison [DP12, ADDP19b] and committee coordination [BDP16] are the two synchronization problems I have investigated. I have worked a lot on self-organizing tasks, mainly spanning tree construction [DKKT10, ADD18, DIJ19] and clustering [DLD+13, DDH+16, DDL19]. Finally, I have worked on many resource allocation problems including mutual exclusion [CDV09a, DDNT10], ℓ-exclusion [CDDL15], k-out-of-ℓ exclusion [DDHL11], and local resource allocation [ADD17b].

1.2 Fault Tolerance

1.2.1 The Need of Fault Tolerance in Distributed Systems

Modern networks are often large-scale and so geographically spread. For example, there were around 17.6 billions of connected computers on the Internet in 2016, and this number is still growing on, *e.g.*, with the advent of the Internet of Things. Moreover, they are made of heterogeneous entities (laptops, desktops, smartphones, watches, ...) that are mass-produced at a low cost, and consequently unreliable. Now, when the size of a network increases, it becomes more exposed to the failure of some of its components. In fact, in such large networks, it is not reasonable to assume that no fault will occur, even during only a couple of hours. Moreover, the availability of wireless communications has tremendously increased in recent years. Such an evolution brings new usages, but also new issues. For example, wireless communications are typically unreliable due to radio frequency interference, signal attenuation, Knife-edge diffraction, ... Moreover, energy of smart wireless connected objects (*e.g.*, smartphone, tablet, sensor) is usually provided by batteries of limited capacity making them prone to crash failures. Hence, faults are unavoidable during the lifetime of today's networks. Moreover, networks are ubiquitous in our everyday life, and people are increasingly dependent on them. So, any disruption of their services, even temporary, is undesirable and the consequences may be even severe in case of critical applications, *e.g.*, healthcare, air traffic control. Now, due to their large-scale, human intervention to repair them is at least undesirable and even sometimes impossible. As a result, modern distributed systems should achieve *fault tolerance*, *i.e.*, they should automatically withstand (*a.k.a.*, *masking* approach) or at least recover from (*a.k.a.*, *non-masking* approach) failure events. In other words, the possibility of failure should be directly considered in the design of the distributed algorithms them-selves.

1.2.2 A Taxonomy of Faults

There is a tight distinction between fault, error, and failure. The relationship between those three terms can be summarized as follows. A fault causes an error that leads to a failure.

The *failure* of a system component (link or process) means that the behavior of the component is not correct w.r.t. its specification. For example, a process may stop executing its program, or a link may lose some messages. A process is said to be *correct* (resp. a link is said to be *reliable*) if it never experiences any failure.

Notice that a link is reliable if it satisfies the following three properties:

No Creation: Every message received by a process p from a process q has been previously sent by q to p.

No Duplication: Every message is received at most once.

No Loss: Every sent message is delivered to its destination process within finite time.

An *error* is a state of the system that may lead to a failure. It can be a software error (*e.g.*, division by zero, non-initialized pointer), or a physical error (*e.g.*, disconnected wire, turned-off CPU, a wireless connection drop).

A *fault* is the event that has caused an error, *i.e.*, programming faults for software errors, or a physical event (*e.g.*, power outage, disturbance in the environment of the system) for physical errors.

Here, we will only consider physical faults. A fault can be classified according to the following criteria:

Localization: the network component (process or link) hit by the fault.

Cause: whether the fault is *benign*, *i.e.*, unintentional (*e.g.*, material obsolescence), or *malign*, *i.e.*, due to malicious attacks coming from outside (*e.g.*, a virus).

Duration: the duration is either *permanent*, *i.e.*, greater than the remaining time of execution, or *temporary*.

Temporary faults are either *transient* or *intermittent*. Transient and intermittent faults occur at an unpredictable time, but do not result in a permanent hardware damage. In both cases, network components (processes or links) affected by such faults temporarily deviate from their specifications, *e.g.*, some bits in a process local memory can be unexpectedly flipped, some messages in a link may be lost, reordered, duplicated, or even corrupted. The tight difference between transient and intermittent lies on their frequency, low in the transient case and high in the intermittent case (further explanations will be given hereafter).

Detectability: a fault is detectable if its effect on the state of a process allows this latter to be eventually aware of it.

Given these criteria, several fault patterns are classically studied in the literature. Below, we only list a few of them.

Process crash. A process *crashes* when it definitely stops executing its program.

Lossy link. A link is *lossy* when some messages sent through this link vanish without being received. This kind of fault is also referred to as *omission fault* in the literature. Notice that, solutions withstanding message losses often assume that all or part of links are *fair lossy* [BCT96]: in a fair lossy (directed) link (p,q), if infinitely many messages are sent by process p to process q, then infinitely many messages are delivered to q, *i.e.*, infinitely many messages are received by q, provided that q regularly invokes its reception primitive to receive these messages.[2] Notice that the fair loss of messages is a kind of *intermittent fault*.

Byzantine process. A process is *Byzantine* when its behavior is arbitrary, in particular it may no more correspond to the code of its program. Byzantine processes typically model processes infected by some viruses.

[2] An example of fair lossy link is a link where each one in two consecutive messages is lost.

Transient fault. Network components affected by *transient faults* temporarily deviate from their specifications. As a result, after a finite number of transient faults, the configuration of a distributed system may be arbitrary, *i.e.*, variables in process memories may have arbitrary values taken in their respective definition domains (*e.g.*, a Boolean variable is either true or false) and communication links may contain a finite number of arbitrary valued messages. However and as opposed to intermittent faults, after faults cease, we can expect a sufficiently large time window without any fault so that the system recovers and then exhibits a correct behavior for a long time.

Notice that the tight difference between transient and intermittent faults is not quantitatively defined. Indeed, this difference is rather a point of view on the way such faults should be handled: following a non-masking approach for the transient case, while the intermittent case is usually treated in a masking manner.

1.2.3 On the Difficulty of Achieving Fault Tolerance in Distributed Systems

The difficulty of achieving fault tolerance in distributed systems often relies on their asynchronous aspect. More generally, assuming several sources of uncertainty in a distributed system (those sources include, but are not limited to, faults, asynchrony, mobility, anonymity) makes the problems harder, even sometimes impossible, to solve in distributed settings. As an illustrative example, consider the (binary) *consensus* problem and assume a fully-connected distributed system equipped with reliable bidirectional communication links.

If the system is asynchronous but does not endure any process crash, then the solution for the consensus problem is trivial: every process broadcasts its input value and then collects input values from all other processes; finally, all processes holding the same (full) knowledge can decide safely using any deterministic rule that does not violate the validity property.

Consider now systems that are prone to process crashes. First of all, the consensus specification should be adapted to take the possibility of failures into account. Namely, the termination requirement should now only concern the correct processes (*i.e.*, the processes that never crash). Then, if this unsafe system is synchronous, still the consensus can be easily solved. For example, assuming the maximum number crash is f,[3] the *FloodSet* algorithm [Lyn96] consists in $f+1$ rounds during which all processes broadcast each time all inputs they know (together with the identifier of the associated process). Again, at the end of the $f+1$ rounds, correct processes have the same information (in particular, the input values of all correct processes), and can decide exactly as in the previous case.

Now, if the system is asynchronous and may suffer from crash failures, then the consensus problem is impossible to solve [FLP85]. More precisely, Fischer, Lynch, and Paterson demonstrated in 1985 that there is no deterministic algorithm solving the (binary) consensus in an asynchronous fully-connected distributed system yet equipped with reliable links and where only one process may crash. Roughly speaking, this impossibility mainly relies on (1) the fact that in an asynchronous crash-prone system, processes cannot distinguish an arbitrary slow process from a crashed one; and (2) for every deterministic decision rules satisfying validity, there always exists an input assignment such that the input value of a single process can tip the decision.

This latter result is the cornerstone in the distributed fault tolerant research. Indeed, it shows that in very favorable scenarios (*e.g.*, fully connected network, reliable links, and at most one process crash), there is no way to agree in an asynchronous network, even when the agreement involves only two possible decision values. Now, most of the distributed tasks (*e.g.*, detection termination, barrier synchronization, commit in

[3] There is no condition on f, it can be even the number of processes, or an upper bound on it. However, the time complexity of the solution depends on the choice for f.

distributed databases, ...) require agreement. By the way, many important distributed problems, such as atomic broadcast [CT96], have been proven equivalent to the consensus, meaning that using the former we can solve the latter and *vice versa*.

Beyond that, this fundamental result shows that issues in fault tolerant distributed algorithms mainly come from the following sources: faults, asynchronism, determinism, and decision. In fact, if one removes or weakens one of these sources, then the consensus problem becomes solvable.

Indeed, if appropriate assumptions on faults are made or processes can access pertinent information on faults, then the consensus becomes solvable. For example, Fischer, Lynch, and Paterson [FLP85] propose an algorithm solving the consensus if a majority of processes are correct and no process dies during the execution, *i.e.*, processes are either correct, or initially dead. Chandra and Toueg [CT96] solve the consensus by assuming each process can access an oracle, called *failure detector*, that gives maybe partial and not completely accurate information about crashes.

Consensus is also solvable in some systems that are assumed to be not fully asynchronous, *a.k.a.*, in *partially synchronous* systems; see [DLS88].

Other way to circumvent the impossibility is to weaken the problem specification itself. For example, Kuhn *et al.* [KMO11] define the *eventual consensus*, where the integrity property is removed: each process can change several times its decision, but the convergence to a common decision value is required. Ben-Or [BO83] proposes a probabilistic solution for the consensus following the Las Vegas approach, *i.e.*, termination is only guaranteed with probability one.[4] In a same line of research, *self-stabilization* [Dij74, ADDP19a] weakens the achieved specification. Indeed, as we will see, self-stabilization is a non-masking approach, *i.e.*, it does not aim at hiding effects of faults, rather it authorizes the system to temporarily deviate from its intended specification.

1.3 Self-stabilization

1.3.1 Intuitive Understanding: the Collatz Conjecture

The concept of *self-stabilization* can be intuitively understood using the following parable.[5] Assign an arbitrary positive integer value to a variable u and apply the following (sequential) algorithm, noted `A` in the sequel:

```
1: while true do
2:     print u
3:     if u is even then
4:         u ← u/2
5:     else
6:         u ← 3 × u + 1
7:     end if
8: end while
```

For example, initializing u to 12, we obtain the infinite sequence:

$$s = 12, 6, 3, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1 \ldots$$

Remark that s is composed of a finite prefix followed by the infinite repeating suffix $(4, 2, 1)^{\omega}$. Try another initial value, say 29. We obtain the following infinite sequence:

[4] *I.e.*, *almost surely*, meaning that there may be non-terminating executions, but the probability that the execution does not terminate is 0.

[5] This parable has been introduced first in a book I have co-authored with several colleagues [ADDP19a].

$$s' = 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1 \ldots$$

Again, $(4, 2, 1)^\omega$ is an infinite suffix of s'.

Actually, until now, such a convergence phenomenon has been observed for every tested initial (positive integer) value. But, there is still no proof showing the ineluctable convergence of the process regardless of which positive integer is chosen initially: this open problem is known as the *Collatz conjecture* (or *Syracuse problem*).

Define the current configuration of A to be the value of variable u. So, the set of all possible configurations of A is $\mathbb{N}^*$, the set of natural integers. Divide the possible configurations of A into two sets: the set of *legitimate configurations* $\{1,2,4\}$ and the set of *illegitimate configurations* (*i.e.*, non legitimate) $\mathbb{N}^* \setminus \{1,2,4\}$; see the Venn diagram in Figure 1.2.

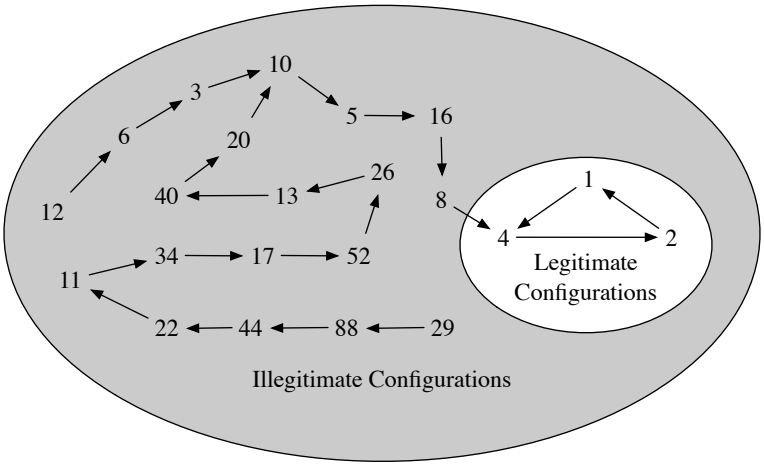


Figure 1.2: Configurations of A.

Then, the Collatz conjecture can be reformulated as follows:

Is Algorithm A *self-stabilizing?*

Indeed, intuitively, a *self-stabilizing* algorithm, regardless of its initial configuration, reaches within finite time a legitimate configuration (*Convergence* property) from which it achieves a specific behavior (*Correctness* property). Such a specific behavior often includes, or even simply consists in (like in the present example), the actual *closure* of the set of legitimate configurations.

1.3.2 A Definition

In the original paper of Dijkstra [Dij73, Dij74], self-stabilization is defined rather informally; moreover the proposed definition is not that general (*e.g.*, the definition is restricted to sequential systems):

" We call the system self-stabilizing if and only if, regardless of the initial state and regardless of the privilege selected each time for the next move, at least one privilege will always be present and the system is guaranteed to find itself in a legitimate state after a finite number of moves."

Notice that, along the years, several close, yet non-equivalent, definitions of self-stabilization have been used in the literature; see, *e.g.*, [Dol00, Gho14]. Until now, there is no consensus in the community on a reference definition.

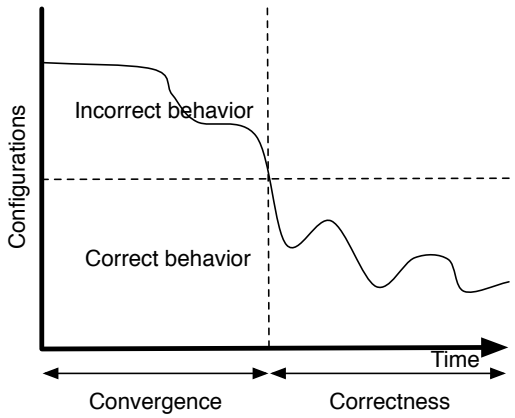


Figure 1.3: Self-stabilization.

My understanding of the notion of self-stabilization is the following: An algorithm is self-stabilizing in a given system (properly defined in terms of interconnected networks, level of anonymity, level of asynchrony, *etc.*) if, regardless of the initial configuration, every execution automatically (*i.e.*, without any external, *e.g.*, human, intervention) reaches within finite time a point from which the behavior is correct (*i.e.* satisfy the intended specification) whatever will be the future. This definition is actually similar to the one given in the reference book of Dolev [Dol00]. This latter is split into two properties: the *convergence* and *correctness* properties; see Figure 1.3.

In more detail, consider a distributed algorithm `A` deployed on a given system *S*. An *execution* of `A` in *S* is a sequence of configurations that can occur, starting from an arbitrary configuration. A specification can be characterized by a predicate *SP* defined over all possible executions. Then, `A` is *self-stabilizing w.r.t. SP* in *S* if there exists a (non-empty) subset of so-called *legitimate*[6] configurations such that:

- every execution of `A` in *S* contains a legitimate configuration (*convergence*); and
- every execution (and so every suffix of execution) starting from a legitimate configuration satisfies *SP* (*correctness*).

It is worth noticing that this definition does not include the notion of *closure*, *i.e.*, the fact the only legitimate configurations are reachable from a legitimate one. Now, when dealing with high-level models (such as the atomic-state model), closure is most of the time present in definitions of self-stabilization; see, *e.g.*, [Gho14]. However, in the more practical message passing model, closure is usually simply given up; see, *e.g.*, [APVD94, DIM97a, Var00]. Even if this absence is never motivated, this may be explained by the lack of functional significance of the closure property compared to the convergence and correctness properties. Closure is rather, in my opinion, a nice property that often helps to write elegant, and so simpler, proofs. Moreover, for me, closure is too restrictive since, for example, it requires to only consider suffix-closed specifications; a specification *SP* being *suffix-closed* if every suffix of any execution satisfying *SP* also satisfies *SP*. Now, for several problems, defining a suffix-closed specification is difficult, even sometimes impossible. Consider, for example, the *data-link* problem in message passing, which classically aims at providing two high-level primitives (one for sending and the other for receiving) allowing to reliably (*i.e.*,

[6]Legitimate configurations are called *safe* configurations in [Dol00], but I prefer to use the word *legitimate* which is more commonly used in the area.

without loss, duplication, nor creation) transmit, in a FIFO manner, data packets from a sender to a receiver over an unreliable channel. Then, this data-link problem cannot be defined using a suffix-closed specification. Indeed, one can always select a suffix s of a correct execution where a message has been sent before the beginning of s but is still under transmission in s. Hence, there is a message creation in the execution s, *i.e.*, s does not satisfy the data-link specification.

1.3.3 The Stabilization Time

A self-stabilizing algorithm ensures that, starting from an arbitrary configuration, every possible execution contains a *finite* prefix made of illegitimate configurations only, called the *stabilization phase*. Now, the correct behavior of the system is only guaranteed from a legitimate configuration. So, the primary goal of a self-stabilizing designer is to reduce the size of the stabilization phase. In other words, reducing the *stabilization time*, *i.e.*, the time before reaching the first legitimate configuration, is the major time complexity measure used in self-stabilization.

The set of legitimate configurations is defined by the algorithm designer. Now, to ease the proofs and with few notable exceptions (see, *e.g.*, the first Dijkstra's token ring algorithm [Dij73, Dij74, ADDP19a]), this set is often chosen as a very particular strict subset of configurations from which the problem specification is achieved. Hence, this choice may have a non-negligible impact of the stabilization time (this question is still open).

We now discuss the advantages and drawbacks of the self-stabilizing approach.

1.3.4 Major Advantage: Fault Tolerance

Self-stabilization is mainly advertised as a general approach to design distributed systems tolerating *any finite* number of *transient faults*. Now, the definition of self-stabilization does not directly refer to the possibility of (transient) faults. Consequently, modeling and proving a self-stabilizing system do not involve any failure pattern; see Figure 1.3. Actually, this is mainly due to the fact that, in contrast with most of existing fault tolerance (*a.k.a.*, *robust*) proposals, self-stabilization is a non-masking fault tolerance approach. As a result, the faults are not directly treated, but rather their consequences. Furthermore, convergence is guaranteed only if there is a sufficiently large time window without any fault; see Figure 1.4. Hence, in the formal definition and so the proofs, the initial point of observation (referred to as the *initial configuration*) is considered to be after the occurrence of the "last" faults and there is no fault model in the literal sense. That is, the system is studied starting from an arbitrary configuration reached due to the occurrence of some transient faults, but from which it is assumed that *no fault will ever occur*.

Notice that the definition implicitly assumes that faults do not alter the code of the algorithm. This assumption is justified in [Dol00] by the following two arguments. The code of a self-stabilizing algorithm can be hardwired in a ROM (Read Only Memory) which cannot be corrupted. Another solution consists of saving a duplication of the code in trusted long-term memory, such as a hard disk, and regularly reloading the code from that trusted memory.

Finally, assuming that transient faults may lead to completely arbitrary configurations may seem to be too strong. However, Varghese and Jayaram [VJ00] have shown that, for example, unreliable links and process crash-recoveries[7] can drive a distributed system to an arbitrary configuration. On the other hand, considering an arbitrary initial configuration implies no hypothesis on the nature or extent of transient faults that could hit the system. Therefore, self-stabilization allows a system to recover from the effects of those faults in a unified manner.

[7] The crash-recovery of a process consists in a sudden reinitialization of the process to its predefined initial state.

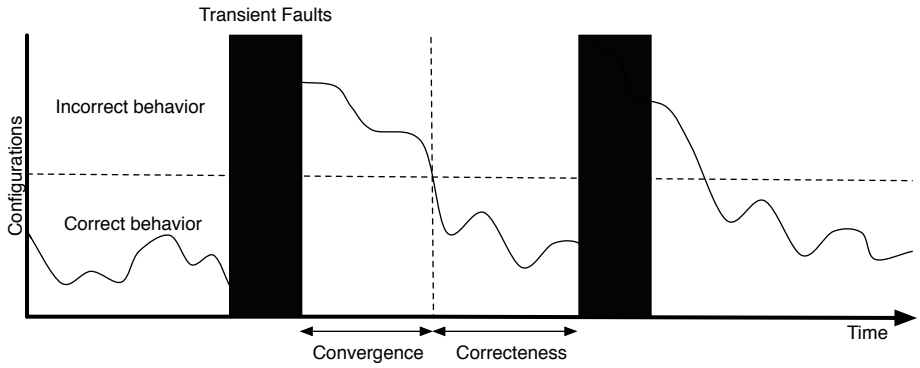


Figure 1.4: Fault tolerance of self-stabilizing systems.

1.3.5 A Lightweight Approach for Fault Tolerance

The overhead of self-stabilizing algorithms compared to non fault-tolerant algorithms is observable regarding execution time, memory requirement, and exchanged information. It is worth noticing that for many problems, the overhead of the state-of-the-art self-stabilizing algorithms is asymptotically negligible; see [ADDP19a]. Actually, self-stabilization is often considered as a *lightweight* fault tolerance technique compared with the classical robust approach [Tix06]. This is probably due to the fact that self-stabilization follows an *optimistic* approach, while robust algorithms follow a *pessimistic* one [Tel01]. The goal of the *pessimistic* approach is to prevent the system from deviating from its specification, in particular in presence of faulty behaviors, by preceding each step with sufficient checks in order to validate it. By contrast, the *optimistic* approach consists in never suspecting the occurrence of non-expected events such as faults, and thus may cause inconsistent behaviors to happen, but guarantees that any deviation from the specification is temporary. The lightweight aspect of self-stabilization makes it very attractive for low-capability and resource-constrained networks such as wireless sensor networks (WSNs) [HT04].

1.3.6 Other Advantages

Performing a consistent initialization is a complex and critical synchronization task in distributed systems. For example, a large-scale network (*e.g.*, Internet) involves numerous processes that may be geographically far from each other and this makes consistency of initialization particularly hard to obtain. So, self-stabilizing algorithms are very desirable for such systems because self-stabilization requires no kind of initialization.

Furthermore, even if self-stabilizing algorithms are mostly designed for static topologies, those dedicated to arbitrary network topologies tolerate, up to a certain extent, some topological changes (*i.e.*, the addition or the removal of communication links or nodes). Precisely, if topological changes are eventually detected locally at involved processes and if the frequency of such events is low enough, then they can be considered as transient faults.

These two latter advantages make self-stabilization naturally suited for *autonomic computing*. The concept of autonomic computing was first used by IBM in 2001 to describe computing systems that are said to be *self-managing* [HM08, KC03]. The spirit of autonomic computing is to capture an extensive collection of concepts related to *self-* capabilities*, *e.g.*, self-organization, self-healing, self-configuration, self-management, self-optimization, self-adaptiveness, or self-repair. Roughly speaking, autonomic computing gathers all techniques allowing a distributed system to adapt to unpredictable changes while hiding intrinsic complexity to operators

and users. Therefore, self-stabilization can be seen as a theoretically founded alternative for the design of autonomic systems.

Finally, the fact that self-stabilizing algorithms never deadlock despite the arbitrary configuration of the system makes them easy to compose [Her92b, Tel01]. Consider, for example, two self-stabilizing algorithms `A` and `B` such that `B` takes the output of `A` as input. `A` and `B` can be executed in parallel since eventually `A` will provide a correct input to `B` and from that point, the actual convergence of `B` will be guaranteed.[8]

1.3.7 Relative Drawbacks and Alternatives

As mentioned earlier, self-stabilization is a non-masking fault tolerance approach. That is, after transient faults cease, there is a finite period of time, called the *stabilization phase*, during which the safety properties of the system may be violated (recall that legitimate configurations are a well-chosen subset of configurations from which the problem specification is achieved). Loss of safety is clearly the main drawback of the self-stabilizing approach. This is why reducing the stabilization time definitely is the primary target of algorithm designers. Another important line of research has been to propose stronger forms of self-stabilization to mitigate the effects of the loss of safety during the stabilization phase. In other words, all these variants of self-stabilization offer extra safety guarantees. For instance, the *fault-containment* [GGHP07] approach additionally ensures that when few (transient) faults hit the system, the faults are both spatially and temporally contained. *Time-adaptive* self-stabilization [KPS99] guarantees a stabilization time linear in the number of faults f, if f does not exceed a given threshold. *Snap-stabilizing* algorithms [CDD$^+$16] recover a correct behavior immediately after transient faults cease. A short survey of strong variants of self-stabilization will be presented in Section 1.5.

Self-stabilizing algorithms are specifically designed to handle transient failures. Consequently, they are not inherently suited for other failure patterns, *a.k.a.*, intermittent failures and permanent failures. Actually, most of existing self-stabilizing solutions become totally ineffective in case of permanent failures such as process crashes, or Byzantine failures. However, notice that several self-stabilizing algorithms also support intermittent failures, such as frequent lost, duplication, or reordering of messages, *e.g.*, [DT02, DDT06, DDLV17]. Moreover, strong forms of self-stabilization have been introduced to cope with process crashes, *e.g.*, *fault-tolerant self-stabilization* [BK97, DDF10], and Byzantine faults, *e.g.*, *strict stabilization* [NA02a, DMT15].

In self-stabilizing systems, processes cannot locally decide whether the system has globally converged, except for few trivial specifications. Indeed, assume a situation where a process p truly decides, using available local information, that the system has converged. Then, it is almost always possible to build a configuration in which p has access to exactly the same local information but where the system has still not stabilized (*e.g.*, by simply modifying the local state of a process 2-hops away from p). Consequently, p cannot distinguish between the two situations. In the second one, p will take a wrong decision. As a consequence, usually all processes should run their local algorithm forever in order to permanently check whether or not they should update their local state after learning fresh information, *e.g.*, message passing methods for self-stabilization usually require the use of *heartbeat* messages (*i.e.*, control messages that are periodically sent) [APSV91, DDT06]; algorithms implementing such a heartbeat method are said to be *proactive* in the literature. However, notice that Arora and Nesterenko [AN05] mitigated this problem, since they proposed non-proactive, *a.k.a. reactive*, self-stabilizing solutions for non-trivial problems, such as *mutual exclusion*, where message exchanges eventually stop in absence of further input modification (*e.g.*, in absence of any further request). Yet, their method cannot be generalized since they also show that many specifications, like

[8] This composition technique, introduced by Herman in [Her92b], is called the *collateral composition*.

the usual *leader election* specification, only admit self-stabilizing proactive solutions.

Finally, additional assumptions may be necessary to obtain self-stabilization. For example, a non self-stabilizing data-link protocol can be implemented using only two sequence values [Lyn68]. This latter algorithm, known as the *Alternating Bit Protocol*, works using bounded (actually small) local process memories in message passing where links are assumed to be bidirectional, fair lossy, yet of unbounded capacity. Now, Gouda and Multari [GM91] show that it is impossible to implement a (deterministic) self-stabilizing data-link protocol under such assumptions. However, they also show that the problem becomes self-stabilizingly solvable still with unbounded capacity links, yet assuming infinite process memories. In [Var00], Varghese circumvents the impossibility by proposing a self-stabilizing (deterministic) algorithm, using bounded local process memories, for the case where a bound on link capacity is known by all processes.

More generally, impossibility results have motivated the introduction of weak versions of self-stabilization. For example, like for the non fault-tolerant algorithmics, many impossibilities in self-stabilization are due to anonymity, *i.e.*, many classical problems, such as leader election in trees [YK96] or token passing in directed rings [Her90], have no self-stabilizing solution in case of process-anonymity.[9] Now, those problems can be solved, for instance, by both a *weak stabilizing* algorithm (see [DTY15]) and a *probabilistically self-stabilizing* algorithm (see [DIM97b] and [IJ90][10]). In more detail, both weak and probabilistic self-stabilization require a convergence property that is weaker than that of self-stabilization. Indeed, weak stabilization stipulates that starting from any initial configuration, there exists an execution that eventually reaches a point from which its behavior is correct; and probabilistic self-stabilization requires that, regardless of the initial configuration, every execution converges with probability one to a point from which its behavior is correct. A short survey on generalizations of self-stabilization will be presented in Section 1.5.

1.4 Computation Models used in Self-stabilization

In distributed computing, both correct operation and complexity of distributed algorithms are commonly established using paper-and-pencil proofs. An important criteria is then the computational model in which algorithms are written. So far, three main models have been extensively studied in the self-stabilizing area (from the weakest to the strongest): the (classical) *message passing model* [KP93], the *register model* (also called the locally shared memory model with read/write atomicity) [DIM93, Dol00], and the *atomic-state model* (also called locally shared memory model with composite atomicity) [Dij73, Dij74].

1.4.1 Message Passing Model

The message passing model is the closest from real-life networks. In this model, neighboring processes can exchange information through messages sent in an intermediate communication medium, called *link* or *channel*. This latter allows to formalize a physical wire in a wired network, for example. To use such a communication medium, each process has two communication primitives, one for sending messages in a link, the other for receiving messages from a link.

In message passing systems, time complexity is usually evaluated in terms of *time units*, also called *asynchronous rounds*. The overall idea is that the process execution time is negligible compared to the message transmission time. First, for separation of concerns, only executions where all links are reliable

[9] *I.e.*, all processes have the same program with the same set of possible states. In particular, they have no local parameter (such as an identity) permitting them to be differentiated. In such systems, any two processes cannot be distinguished, unless they have different local degrees.

[10] Actually, solutions in [DIM97b] and [IJ90] work under even more general settings: anonymous arbitrary connected networks.

are considered in the time complexity analysis. This simplification is justified by the fact that measuring time complexity under unreliable links does not only evaluate the performance of the algorithm, but rather the performance of the whole system, including the network. Now, a time complexity bound should be independent of system-specific parameters. Then, each message transmission is assumed to last at most one time unit and each process step is assumed to last zero time unit. The idea is that time complexity should measure the execution time of the algorithm according to the slowest messages: the execution is normalized in such a way that the longest message delay (*i.e.*, the transmission of the message followed by its processing at the receiving process) becomes one unit of time.

Notice that I have investigated self-stabilization in this model in several works; see [DDNT10, DDHL11, DDLV17].

The register and atomic-state models presented below are actually abstractions of the message passing model where message exchanges between neighbors are replaced by a read access to the state of the neighbors. The register model and the atomic-state model differ by their atomicity assumption.

1.4.2 Register Model

In the register model, each process holds some communication registers, that are shared with its neighbors, and (maybe) some internal, *i.e.*, non-shared, variables. An atomic step consists of one process making internal computation followed by either a read or write action on a communication register.[11] Precisely, a read action allows the process to get the value of one register owned by a neighbor, while a write action allows the process to update one of its own registers. Notice also that executions in the register model are assumed to be fair: if infinitely often a process has a step to execute, then the process executes infinitely many steps.

In the register model, time complexity is evaluated in terms of *rounds*, which are essentially the transposition to shared memory models of the notion of time units (or asynchronous rounds) used in message passing systems. The first round of an execution is its minimum prefix containing at least one step per process. The second round of the execution is the first round of the execution suffix starting from the last configuration of the first round, and so on.

Notice that I have not worked on the register model until now.

1.4.3 Atomic-State Model

The *atomic-state model* has been introduced by Dijkstra in its primary work on self-stabilization [Dij73, Dij74]. Since then, it is the most commonly used model in self-stabilization. This is also the model I have mostly considered in my work; see, *e.g.*, [ACD$^+$17b, AD17, ADD17b, ADDP19b, DDL19]. Consequently, we will make a wider focus on it.

In this model, atomicity is stronger than in the register model. Indeed, each atomic step consists of at least one process (maybe several) reading its state and that of all its neighbors, and updating its own state.

In more detail, in this model, a distributed algorithm consists of a collection of local algorithms, one per node. The local algorithm of each node p is defined as a finite set of shared registers, simply called *variables*, and a finite set of *actions* (written as guarded commands) to update them. Some of the variables may be constant inputs from the system in which case their values are predefined. A node can read its own variables and that of its neighbors, but can write only to its own (non-constant) variables.

Each action is of the following form: $\langle label \rangle :: \langle guard \rangle \hookrightarrow \langle statement \rangle$. *Labels* are only used to identify actions. A *guard* is a Boolean predicate involving the variables of the node and that of its neighbors.

[11] The fact that executions are sequential in the register model is not a restriction, it is rather a way to simplify the modeling. Indeed, under such an atomicity, any concurrent step can be serialized, *i.e.*, can be simulated by a finite number of sequential steps.

The *statement* is a sequence of assignments on variables of the node. An action can be executed only if its guard evaluates to *true*, in which case, the action is said to be *enabled*. By extension, a node is said to be enabled if at least one of its actions is enabled.

Nodes run their local algorithm by *atomically* executing actions. During an execution, the interleaving between executed actions is *nondeterministically* decided by an adversary called *daemon*, which models the asynchronism of the system. Precisely, an execution is a sequence of configurations, where the system moves from a configuration to another as follows. Assume the current configuration of the system is γ. If no node is enabled in γ, the execution is done and γ is said to be *terminal*. Otherwise, the daemon *activates* a non-empty subset S of nodes that are enabled in γ; then every node in S *atomically* executes one of its action enabled in γ, leading the system to a new configuration γ', and so on. The transition from γ to γ' is called a *step*. Note that several nodes may atomically execute at the same step but there is no default fairness assumption, except progress (at least one node executes an action at each step).

As explained before, asynchronism in the model is captured by the notion of *daemon*, an adversary which decides the interleaving of process executions. The power of a daemon is characterized by its *spreading* and its *fairness* [ADDP19a].

1.4.3.1 Spreading

The spreading restricts the choice of the daemon at each step obliviously, *i.e.*, without taking past actions into account. It is a safety property in the sense of Alpern and Schneider; see [AS85]. Below, I present the four most popular spreading assumptions of the literature.

- A daemon is *central (or sequential)*, noted $\mathbb{C}$, if it activates exactly one process per step.
- A daemon is *locally central*, noted $\mathbb{LC}$, if it activates at least one process (maybe more) at each step, but never activates two neighbors in the same step.
- A daemon is *synchronous*, noted $\mathbb{S}$, if it activates every enabled process at each step.
- A daemon is *distributed*, noted $\mathbb{D}$, if it activates at least one process (maybe more) at each step, *i.e*, it is not restricted in terms of spreading.

1.4.3.2 Fairness

Fairness allows to regulate the relative execution rate of processes by taking past actions into account. It is a liveness property in the sense of Alpern and Schneider; see [AS85]. Below, I present the three most popular fairness assumptions of the literature.

- A daemon is *strongly fair*, noted $\mathbb{SF}$, if it activates infinitely often all processes that are enabled infinitely often.
- A daemon is *weakly fair*, noted $\mathbb{WF}$, if it eventually activates every continuously enabled process.
- An *unfair* daemon, noted $\mathbb{UF}$, has no fairness constraint, *i.e.*, it might never select a process unless it is the only enabled one.

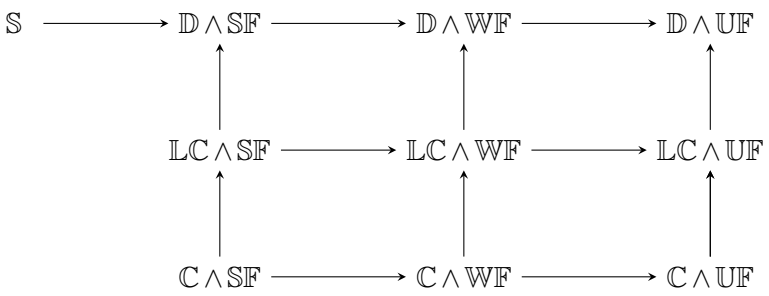


Figure 1.5: Relationships between the main daemons: $D \succeq D'$ if there is a directed path from D to D'.

1.4.3.3 Hierarchy of Daemons

A daemon D is said to be *stronger* than a daemon D' if for every execution e, we have $D(e) \Rightarrow D'(e)$; in this case, D' is said to be *weaker* than D. We denote by $D \succeq D'$ (resp. $D' \preceq D$) the fact that D is stronger that D' (resp. D' is weaker than D). Note that if D' is weaker than D, then every algorithm that is self-stabilizing assuming D' is also self-stabilizing assuming D. In contrast, every problem that has no self-stabilizing solution under D has no solution under D' too.

By definition, the *distributed unfair daemon* is the most general daemon of the model. Hence, solutions stabilizing under such an assumption are highly desirable, because they work under any other daemon assumption. Other trivial relationships between classical daemons are given in Figure 1.5. Notice in particular that, following the literature, the synchronous daemon appears in the figure without being coupled with any fairness assumption. This is justified by the fact that $\mathbb{S}$ implies every fairness property. However, we consider it as a spreading property because it is a safety by definition: to show that an execution is not synchronous, it is sufficient to exhibit a finite prefix containing a non-synchronous step.

Many other daemons have been defined in the literature; see [DT11] for a detailed survey.

1.4.3.4 Time Complexity Units

Three main units of measurement are used in the atomic-state model: the number of *rounds*, *moves*, and *(atomic) steps*.

The complexity in *rounds* [CDPV02] evaluates the execution time according to the speed of the slowest processes. Essentially, it is the adaptation to the atomic-state model of the notion of rounds in the register model. The definition of round uses the concept of *neutralization*: a process v is *neutralized* during a step from a configuration γ to a configuration γ', if v is enabled in γ but not in configuration γ', and it is not activated in that step. The neutralization of a process v represents the following situation: at least one neighbor of v changes its state between γ and γ', and this change effectively makes the guard of all actions of v false. Then, the rounds are inductively defined as follows. The first round of an execution $e = \gamma_0\gamma_1\cdots$ is its minimal prefix e' such that every process that is enabled in γ_0 either executes an action or is neutralized during a step of e'. If e' is finite, then the second round of e is the first round of the suffix $\gamma_t\gamma_{t+1}\cdots$ of e starting from the last configuration γ_t of e', and so forth.

The complexity in *moves* captures the amount of computations an algorithm needs. Indeed, we say that a process *moves* in $\gamma \mapsto \gamma'$ when it executes an action in $\gamma \mapsto \gamma'$. So, it is rather a measure of work than a measure of time.

The complexity in *(atomic) steps* essentially captures the same information as the complexity in *moves*.

Indeed, the number of moves and the number of steps are closely related: if an execution e contains x steps, then the number y of moves in e satisfies $x \leq y \leq n \cdot x$, where n is the number of processes. Actually, in most of the literature, upper bounds on step complexity are established by proving upper bounds on the number of moves.

1.4.3.5 Stabilization Time and Daemon

To obtain practical solutions, the designer usually tries to avoid strong assumptions on the daemon, like for example, assuming all executions are synchronous. Now, when the considered daemon does not enforce any bound on the execution time of processes, then the stabilization time in moves can be bounded only if the algorithm works under an unfair daemon. For example, if the daemon is assumed to be *distributed and weakly fair* and the studied algorithm actually requires the weakly fairness assumption to stabilize, then it is possible to construct executions whose convergence is arbitrarily long in terms of atomic steps (and so in moves), meaning that, in such executions, there are processes whose moves do not make the system progress in the convergence. In other words, these latter processes waste computation power and so energy. Such a situation should be therefore prevented, making the unfair daemon more desirable than the weakly fair one. As a matter of fact, if the daemon is assumed to be weakly fair, then by definition each round is finite. Yet in this case the number of steps in a round can be bounded only if the studied algorithm actually works without this fairness assumption, *i.e.*, under (at least) an unfair daemon with the same spreading assumption. On the other hand, an algorithm can be proven under the distributed unfair daemon using the following proof scheme: it is first shown assuming a distributed weakly fair daemon; then, it remains to show that, under the distributed unfair daemon, each round contains a finite number of steps [CDPV06].

1.4.4 Model Conversion

General methods for converting an algorithm from one model to a weaker one, while preserving the self-stabilization, exist; see, *e.g.*, [Dol00].

As a matter of facts, using any data-link protocol, the conversion from the register to the message passing model is straightforward. Moreover, the overhead in time and space is low; see [ADDP19a].

The conversion from the atomic-state to the register model is more problematic. Indeed, the only existing general solution [Dol00] ensures the strong atomicity by enforcing sequentiality, making the time overhead huge. Nevertheless, the overhead in space is still low: at each process p, the memory requirement is multiple by a factor of $O(\delta_p)$, where δ_p is the number of p's neighbors.

To the best of my knowledge, there is still no general method to directly (*i.e.*, without using the transformation from the register to the message passing model) convert any self-stabilizing algorithm written in the atomic-state into the message passing model. However, some studies, *e.g.*, [Var00, DDT06, ADDP19a], show that several solutions to important problems (*e.g.*, spanning tree construction, PIF in general rooted networks) can be efficiently brought from the atomic-state to the message passing model.

1.5 Variants of Self-stabilization

Many variants of self-stabilization have been introduced over the years; see, *e.g.*, the two surveys I have co-authored [DPV11a, DPV11b]. I now only review those I have worked on. Notice also that, with some colleagues, I also have introduced some variants of self-stabilization. These latter will be presented in Chapter 4.

1.5.1 Weak Forms of Self-stabilization

Weak forms of self-stabilization have been introduced for two main reasons:

Circumventing impossibility results. For example, in [BDGPBT09], we have extended the impossibility results on the self-stabilizing vertex-coloring in the atomic-state model from anonymous bidirectional networks to anonymous unidirectional networks. Then, we have circumvented this impossibility by proposing a probabilistic self-stabilizing vertex coloring in the atomic-state model for unidirectional anonymous networks.

Reducing the cost (often in space) of solutions. For example, in message passing, any (deterministic) self-stabilizing solution to the data-link problem requires an infinite memory if the link capacity is unbounded [GM91, DIM97a], while pseudo-stabilizing [BGM93] or probabilistically self-stabilizing solutions [AB93] to that problem have a typically small memory requirement.

Below, I present some weak forms of self-stabilization I have worked on. I will also briefly summarize my related contributions.

1.5.1.1 Pseudo-stabilization

Pseudo-stabilization has been introduced by Burns *et al.* [BGM93] in 1993. Compared to self-stabilization, pseudo-stabilization relaxes the notion of "point" in the execution from which the behavior is correct: every execution simply has a suffix that exhibits correct behavior. Hence, pseudo-stabilization consists in a convergence property that is similar to that of self-stabilization but a weaker correctness property.

The difference between self- and pseudo- stabilization may seem very tight. But actually it is fundamental. For example, when the link capacity is assumed to be unbounded, any self-stabilizing data-link protocol requires an infinite memory [GM91], while a pseudo-stabilizing solution for that problem can be implemented using a typically small memory at each involved process [BGM93].

To clearly understand this difference, we now study an illustrative example proposed in [BGM93]. Consider a system with only two possible states a and b, where there are only three possible transitions:

$$a \to a,\ a \to b, \text{ and } b \to b$$
(*n.b.*, the nondeterminism may be due to the asynchronism).

Then, this system is trivially pseudo-stabilizing for the specification "the execution either only contains a-states, or only contains b-states". Indeed, every execution has a suffix of the form a^ω or b^ω. Now, this system is not self-stabilizing for the same specification: indeed, no matter the choice of the set of legitimate states (either $\{a\}$, $\{b\}$, or $\{a,b\}$), one of the two properties of self-stabilization is not satisfied. If one selects $\{a\}$ as set of legitimate states, then the unique execution starting from b, *i.e.*, b^ω, never converges. If one selects $\{b\}$ as set of legitimate states, then a^ω is a possible execution that again never converges. Finally, if one selects $\{a,b\}$, then every execution of the form $a,\ldots,b,b,b,\ldots$ violates the correctness property.

The main drawback of the pseudo-stabilization is maybe the absence of timing guarantees. In other words, there is no bound on the time to reach a suffix satisfying the intended specification in every execution of a pseudo- yet non self- stabilizing system. For instance, in the previous example, when the execution is of the form $a,\ldots,b,b,b,\ldots$, the correct suffix starts from the first "b", yet the number of "a" before that "b" is unbounded. This is is typically due to the asynchronous nature of the system, *cf.* Section 1.1.2. Notice that I have worked on a variant of pseudo-stabilization dedicated to systems prone to process crashes [DDF10], that will be presented in Chapter 4.

1.5.1.2 Probabilistic Self-stabilization

Probabilistic self-stabilization [IJ90, Her90] differs from deterministic self-stabilization by the convergence property, *i.e.*, in probabilistic self-stabilization, the convergence is not certain since it is only guaranteed with probability one.

In [BDGPB+10], we have investigated the vertex-coloring problem in unidirectional anonymous networks in a probabilistic self-stabilizing setting. In particular, we have proposed a self-stabilizing parametric algorithm achieving a trade-off between time and space; see Section 1.6.2.2 (page 28) for more detail.

1.5.1.3 k-Stabilization

The intuition behind *k-stabilization* is that when a few faults hit the system, it should be easier to recover a correct behavior. k-Stabilization is based on the notion of *Hamming distance* [DH95] between configurations: the Hamming distance between two configurations γ and γ' is the number of processes whose state differs between γ and γ'. Hence, a configuration is said to contain k faults if its minimal distance to any legitimate one is k. Then, k-stabilization [BGK98] guarantees the system converges to a legitimate configuration starting from any configuration containing at most k faults (the correctness property remains the same as in self-stabilization).

In [DDLT13], we consider the atomic-state model using the distributed unfair daemon hypothesis. We assume at most k *arbitrary* memory corruptions may hit the system. In this context, we give a leader recovery protocol that recovers a legitimate configuration where a single leader exists. Precisely, if a leader is elected before state corruption, the *same* leader is elected after recovery. The solution we propose works in any anonymous bidirectional, yet oriented, ring of size n, and does *not* require that processes know n, yet the knowledge of k is assumed. If $n \geq 18k+1$, our protocol recovers the leader in $O(k^2)$ rounds using $O(\log k)$ bits per process. Our protocol handles *erratic* faults in the sense that k tolerated memory corruptions do not necessarily occur in the initial configuration. In fact, they may occur in an erratic way after the network has started recovering the leader. In other words, faults that can be handled by our protocol are not only arbitrarily placed, but also arbitrarily timed.

1.5.1.4 Weak Stabilization

Again, *weak stabilization* [Gou01] differs from self-stabilization by its convergence property.[12] Precisely, weak-stabilization stipulates that from any configuration, there should always exist at least one execution that converges to a legitimate configuration.

In [DTY15], we compare the relative expressive power of deterministic self, probabilistic self, and weak stabilization in the atomic-state model. In particular, we show that any *finite state* deterministic weak stabilizing algorithm to solve a problem under the strongly fair scheduler is always a probabilistic self-stabilizing algorithm to solve the same problem under the randomized scheduler. Unfortunately, this good property does not hold in general for *infinite state* algorithms. We however show that for some classes of infinite state algorithms, this property remains true.

1.5.2 Strong Forms of Self-stabilization

Among the various strong forms of self-stabilization, *silent self-stabilization* [DGS99] is a bit odd. It has been first introduced in the register model, yet this notion is meaningful independently of the computation model.

[12] Notice that there exist weak forms of self-stabilization that relax the correctness property, *e.g.*, the *loose-stabilization* [SOK+19].

A self-stabilizing algorithm is silent if it converges within finite time to a configuration from which the values of the communication variables used by the algorithm remain fixed. By communication variable, we mean a buffer variable whose values are transmitted to and/or accessed by a neighbor using the communication primitives (usually send/receive or read/write operations). Hence, using a silent algorithm, information exchanged between processes is eventually fixed forever.

Dolev *et al.* [DGS99] justified the interest of silence by the fact that a silent algorithm may utilize less communication operations and communication bandwidth. Moreover, Dolev *et al.* also claim that silence usually implies more simplicity in the algorithm design: as an illustrative example, we give in [DLD$^+$13] a simple condition to compose silent algorithms and we have successfully used this result to easily prove the silent self-stabilization of several complex algorithms; see [DLD$^+$13, DLDR13, DDH$^+$16]. By the way, this simplicity allowed us to certify (*i.e.*, mechanically check) the proof of this result using the proof assistant Coq [ACD19].

Remark that, in the atomic-state model, an algorithm is *silent* if and only if all its executions are finite. Indeed, all variables of a process are actually communication variables since any neighbor can read the whole process state.

Other strong forms of self-stabilization have been introduced for various reasons.

- For example, some approaches, such as *fault-tolerant self-stabilization* [BK97] (ftss) and *strict stabilization* [NA02a], characterize the ability of algorithms to self-stabilize in presence of stronger failure patterns, *i.e.*, transient faults and process crashes for ftss, and Byzantine faults for strict stabilization.

- Some variants, such as *superstabilization* [DH95] and *gradual stabilization* [ADDP19b], aim at efficiently withstanding topological changes (*i.e.*, the addition or the removal of communication links or nodes): see Section 4.3 for details.

  For example, a *superstabilizing algorithm* is self-stabilizing and has two additional properties when transient faults are limited to a single topological change. Indeed, after adding or removing one link or process in the network, a superstabilizing algorithm recovers fast (typically $O(1)$ rounds), and a safety predicate, called a *passage* predicate, should be satisfied all along the stabilization phase.

  Gradual stabilization is a generalization of superstabilization: a gradually stabilizing algorithm recovers its safety little by little after some topological changes.

- Some properties, such as *snap-stabilization* [BDPV99a, BDPV07] and *safe convergence* [KM06], allow to mitigate the effects of the loss of safety during the stabilization phase.

  For example, a *snap-stabilizing* algorithm recovers a correct behavior immediately after transient faults cease. A safely converging algorithm quickly recover an interesting part of its safety before totally recovering.

- Some strong variants of self-stabilization, such as *fault-containment* [GGHP07] and *speculative self-stabilization* [DG13], additionally ensure drastically smaller convergence times in particular favorable cases. For example, a *fault containing* self-stabilizing algorithm ensures that when a few faults hit the system, the faults are both spatially and temporally contained. "Spatially" means that if only a few faults occur, those faults cannot be propagated further than a preset radius around the corrupted processes. "Temporally" means a quick stabilization when a few faults occur.

  A self-stabilizing algorithm is speculative whenever it exhibits significantly better performances in a subset of more probable executions.

Below, I only give details on strong variants of self-stabilization I have worked on. I will also briefly summarize my related contributions.

1.5.2.1 Fault-tolerant Self-stabilization

Self-stabilizing algorithms have been initially introduced to withstand transient faults only. Beauquier and Kekkonen-Monetta [BK97] propose to design algorithms that self-stabilize even when some crashes occur "erratically" in the network, *i.e.*, their frequency may be high or low and so we cannot expect a sufficiently large time-window without faults to help convergence. However, as proven in [BK97], without very strong assertions fault-tolerant self-stabilization is impossible to ensure and concerning the leader election problem we have proven that fault-tolerant self-stabilization is intrinsically impossible to solve. Hence, we consider leader election in [DDF10] message passing systems in which process crashes are static, *i.e.*, we assume that some processes maybe initially crashed, but no more crash occurs along the execution (like for transient faults, we do not directly treat crashes but rather their consequences). However, even under such favorable assumptions, we show that self-stabilization is hard to obtain since it requires strong partial synchrony assumptions. Precisely, we have shown that the existence of a *timely source*[13] and a *fair hub*[14] are not sufficient to make the fault-tolerant self-stabilizing leader election solvable; justifying then the introduction of the *fault-tolerant pseudo-stabilization*, presented in Chapter 4.

1.5.2.2 Self-stabilization with Safe Convergence

The notion of *safe convergence* has been introduced by Kakugawa and Masuzawa in 2006 [KM06]. The main idea behind this concept is the following: for a large class of problems, it is often hard to design self-stabilizing algorithms that guarantee a small stabilization time, even after a few transient faults [GT02]. A long stabilization time is frequently due to the strong specifications that a legitimate configuration must satisfy. Then, the goal of a *safely converging self-stabilizing algorithm* is first to "quickly" (within $O(1)$ rounds is the usual rule) converge to a so-called *feasible* legitimate configuration, where a minimum quality of service is guaranteed. Once such a feasible legitimate configuration is reached, the system continues to converge to an *optimal* legitimate configuration, where more stringent conditions are required. Safe convergence is especially interesting for self-stabilizing algorithms that compute optimized data structures, *e.g.*, minimal dominating sets [KM06], approximation of the minimum weakly connected dominating set [KK07].

In [CDD+15], we have investigated, in the atomic-state model, safely converging silent self-stabilizing solution to the minimal (f,g)-alliance in an arbitrary identified network assuming a distributed unfair daemon. Given two non-negative integer-valued functions on processes, f and g, a subset A of processes is a (f,g)-alliance of the network [DPRS11] if every process p not in A has at least $f(p)$ neighbors in A, and every process p in A has at least $g(p)$ neighbors in A. Then, a (f,g)-alliance A is minimal if no proper subset of A is a (f,g)-alliance of the network. Our contribution has been to propose a silent self-stabilizing algorithm which assumes that every node p has a degree at least $g(p)$ (this hypothesis only aims at ensuring the existence of a solution) and satisfies $f(p) \geq g(p)$, and which is *safely converging* in the sense that starting from any configuration, it first converges to a (not necessarily minimal) (f,g)-alliance in at most four rounds, and then continues to converge to a minimal one in at most $5n+4$ additional rounds, where n is the size of the network. Such a minimal (f,g)-alliance problem is a generalization of several problems that are of interest in distributed computing. For example, the *minimal dominating set* corresponds to a minimal $(1,0)$-alliance since a set is dominating if and only if every node not in the set is incident to at least one member of the set.

[13] A timely source is a process whose all outgoing links achieve time guarantees on message delivery.

[14] A fair hub is a process whose all incoming and outgoing links are reliable.

1.5.2.3 Snap-stabilization

An important part of my work, in particular my PhD thesis [Dev10], deal with snap-stabilization; see, *e.g.*, [CDV09a, DDNT10, BDP16, CDD+16, ADD17b]. Chapter 2 will be dedicated to one of my main contributions to snap-stabilization.

The notion of snap-stabilization is often misunderstood, even in the self-stabilizing community. Consequently, I will now try to clearly explain the basics of this concept, at the risk of being longer.

Snap-stabilization has been introduced by Bui *et al.* in 1999 [BDPV99a, BDPV07]. Intuitively, a *snap-stabilizing* algorithm recovers a correct behavior *immediately* after transient faults cease. Formally, starting from an arbitrary initial configuration (which models the first configuration after the end of the faults, as explained before), a snap-stabilizing system always satisfies its specification.

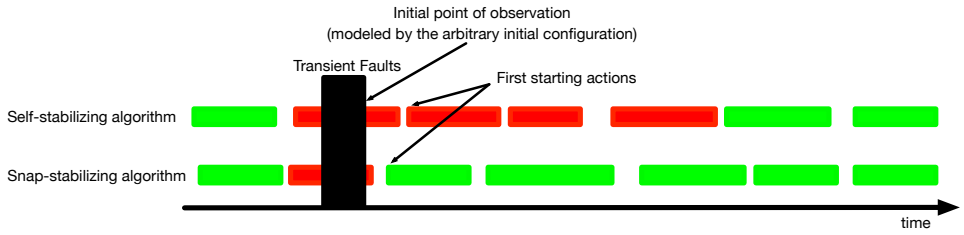


Figure 1.6: Self- versus snap- stabilizing termination detection algorithms. Green queries return a correct answer, red ones return a wrong answer.

Self- *vs.* snap- stabilization. To clearly explain the difference between self- and snap- stabilization, consider, for example, the fundamental problem of *termination detection* and the illustration given in Figure 1.6. In this problem, any process p can be requested (by the application layer) to detect whether some distributed algorithm $\mathcal{X}$ has terminated. More precisely, upon a request a process should initiate a query to know whether $\mathcal{X}$ has terminated, and when p delivers an answer "yes" (resp., "no"), $\mathcal{X}$ "must be terminated" (resp., "may not have terminated"). Let $\mathcal{A}_{\texttt{self}}$ (resp., $\mathcal{A}_{\texttt{snap}}$) be a self-stabilizing (resp., snap-stabilizing) algorithm for detecting the termination of some distributed algorithm $\mathcal{X}$. Let p be any process. If $\mathcal{A}_{\texttt{self}}$ starts from an arbitrary initial configuration and $\mathcal{X}$ eventually terminates, then all we know is that *eventually* (1) only "yes" answers will be computed for all p's queries, and (2) these answers will truly indicate that $\mathcal{X}$ had terminated. However, during the stabilization phase, it is possible that p delivers "yes" answers while $\mathcal{X}$ has not yet terminated. In other words, $\mathcal{A}_{\texttt{self}}$ can compute wrong answers several (but a finite number of) times before finally computing correct answers. In contrast, using $\mathcal{A}_{\texttt{snap}}$, starting from an arbitrary initial configuration, the very first answer delivered by p to any *initiated* query can be trusted to be a correct answer.

It is important to note that snap-stabilizing systems are not insensitive to transient faults. For example, using $\mathcal{A}_{\texttt{snap}}$, if a transient fault occurs between an initiated query and its associated answer, then the answer may not be correct, *i.e.*, a process may deliver "yes" while $\mathcal{X}$ actually has not terminated (in the modeling, starting from the corresponding arbitrary configuration, the process will deliver "yes" to a non-initiated query). However, every answer returned to any query *initiated* after the end of faults (in the modeling, starting from an arbitrary configuration) will be correct. In contrast, $\mathcal{A}_{\texttt{self}}$ just guarantees that only a finite, yet generally unbounded, number of wrong answers will be returned after faults cease.

Static and dynamic specifications. Of course, not all specifications – in particular their safety part — admit snap-stabilizing solutions. For example, in the self-stabilization literature, the safety part of a specification is often defined as a "*static*" condition, *i.e.*, by applying a predicate P over the set of system configurations. Now, with only few notable exception,[15] it is usually impossible to design a snap-stabilizing algorithm for such so-called *static* specifications. Indeed, starting from a configuration that violates the safety predicate P, the specification is violated immediately at the beginning of the execution, regardless of the behavior of the algorithm. Circumventing such an issue can be done using a *dynamic* specification, *i.e.*, a specification that focuses on the notion of a *starting action*, as explained below.

Classically, the correctness of a *non fault-tolerant* algorithm is established by assuming a so-called *safe environment*:

- the system is observed from a pre-defined (well-chosen) initial configuration from which the system is supposed to start;
- moreover, the system is supposed to never suffer from any fault all along its execution.

The design of distributed algorithms for a safe environment distinguishes two types of code [Tel01]: the *spontaneous part* where the code is executed following an implicit external (w.r.t., the algorithm) action called *request* (from an operator or another algorithm) and the *communication part* where the execution of a code is triggered by an information transfer incoming from a neighboring process (*e.g.*, a message reception in message passing). Initialization of any execution of the algorithm is always done by the spontaneous part (if there exist several initiators, they all execute first that part). Then, we call *starting action* the first action of the spontaneous part. Any execution in a safe environment always starts with a starting action. Of course, this is generally not the case for self- or snap- stabilizing algorithms since the first configuration is indeterminate. So, the starting actions may not be the first actions of an execution of a self- or snap- stabilizing algorithm.

We say that a specification is *dynamic* if its safety condition is an implication, where the left part is true when a starting action has been effectively executed before. Therefore, in such a specification, the safety is not only based on the current configuration but rather on the execution prefix that led to that configuration. Actually defining the safety as such an implication should not be considered as a restriction. Indeed, if we consider again the correctness proof of non fault-tolerant algorithms, this implication is implicit: the safety of the specification is proven by showing that between the initialization and termination of the algorithms, a given safety invariant holds. Now, the system exists before the initialization, so the designer implicitly assume that safety cannot be violated between the "real" start of the system and the actual initialization of the algorithm.

Illustrative example: mutual exclusion. We now illustrate the difference between static and dynamic specifications by considering the *mutual exclusion* problem. In this problem, the code of each process is divided into two sections: the *non-critical* and *critical* sections. Processes alternate between these two sections as follows. Initially, a process executes its *non-critical* section. But, sometimes, the application layer *requests* the process to execute its *critical* section. So, a process should enter its critical section in finite time after each request. The time spent into its critical section is assumed to be finite, yet unbounded: each process eventually returns to its *non-critical* section code. Moreover, accesses to critical sections should be managed in such a way that no two critical sections are executed concurrently by two different processes. The transition from the non-critical section to the critical section is implemented by a special code, called the

[15] In [JADT02], Johnen *et al.* propose a snap-stabilizing asynchronous unison algorithm for directed trees whose static safety condition holds in all configurations of the systems.

entry section. Similarly, the transition from the critical section to the non-critical section is implemented in another special code, called the *exit* section. The mutual exclusion problem then consists of the design of the *entry* and *exit* sections.

In the self-stabilization literature, one can find the following static specification of mutual exclusion; see, *e.g.*, [NM02].

Specification 1 (Static Mutual Exclusion)

Safety: *No two processes execute the critical section simultaneously.*

Liveness: *Upon a request, a process enters the critical section in finite time.*

As explained before, it is straightforward to show the impossibility of designing a snap-stabilizing algorithm satisfying Specification 1. Indeed, since several processes may be in their critical section (simultaneously) in the arbitrary initial configuration, the specification can be violated immediately at the beginning of the execution, regardless of the behavior of the algorithm.

Let now propose a dynamic specification of mutual exclusion. Recall that the execution of the entry section is triggered by a request. In other words, every execution of a mutual exclusion algorithm at some process locally repeats the same sequential scheme: Request, Entry Section, Critical Section, and Exit Section. So, in this problem, the starting action corresponds to the first action of the entry section. Hence, a dynamic specification of mutual exclusion can be the following, where we only modify the safety part:

Specification 2 (Dynamic Mutual Exclusion)

Safety: *If a process p enters the critical section, then p executes the critical section alone.*

Liveness: *Upon a request, a process enters the critical section in finite time.*

By contrast with Specification 1, an immediate consequence of Specification 2 is that when the system is an initial configuration where more than one process is executing its critical section, the safety is not immediately violated. Indeed, in such a configuration, no process has ever entered its critical section, *i.e.*, no process has made the transition from the non-critical code to the critical section (using the Entry Section), *i.e.*, no process has executed a starting action. As explained before, in self-/snap- stabilization, the arbitrary initial configuration corresponds to a configuration that can be the result of a finite number of transient faults. So, nothing but transient faults can explain why several processes are executing their critical section in the arbitrary initial configuration. As required for a dynamic specification, the safety condition of Specification 2 is an implication, where the left part is true if a starting action has been effectively executed before.

An interesting question arises from the above discussion: "*Does Specification 2 define the mutual exclusion safety*?" Our answer is yes [CDD$^+$16]. Our justification is the following: in a safe environment, every process which is executing its critical section has previously executed its entry section. So, every algorithm that satisfies Specification 1 also satisfies Specification 2, and *vice versa*. This is actually the general approach we follow in snap-stabilization: in [CDD$^+$16] we define, for every problem, *an equivalence class of specifications*, where the equivalence relation is as follows: for every two specifications $\mathcal{SP}_1$ and $\mathcal{SP}_2$, $\mathcal{SP}_1$ and $\mathcal{SP}_2$ are equivalent if and only if any algorithm satisfying $\mathcal{SP}_1$ in a safe environment also satisfies $\mathcal{SP}_2$ in a safe environment, and *vice versa*. Remark that this equivalence was already implicitly used in fault tolerant distributed algorithmics. For example, the initial termination requirement of the consensus (page 2) has been modified to only consider correct processes when dealing with crash-prone systems (page 6). Now, this modification is valid since all processes are correct in a safe environment.

Another natural question is the following: "*Can we find a snap-stabilizing solution to the mutual exclusion problem*?" Again, the answer is yes. We have proposed in [CDV09a, DDNT10] two algorithms that are snap-stabilizing for Specification 2.

To conclude the discussion about the mutual exclusion problem, starting from an arbitrary initial configuration (or equivalently, after the last transient fault ceases), a snap-stabilizing mutual exclusion algorithm does not guarantee that in this configuration, no two processes can be in the critical section. But, it guarantees that upon any (new) request for a process p to enter its critical section, p will enter the critical section within finite time, and p will not do it unless it can do it safely. Specifically, the snap-stabilizing mutual exclusion algorithm will ensure that no other process is in the critical section before allowing p to enter the critical section. A self-stabilizing mutual exclusion algorithm generally does not provide such a safety property, or simply does not consider this issue.

To summarize, the mutual exclusion example provides a generic approach for writing specifications compatible with snap-stabilization, that can be formulated as follows: *Just recognize the classical starting action in a safe environment, and add it as the condition of the safety*. Consider any distributed algorithm $\mathcal{A}$. If $\mathcal{A}$ is self-stabilizing, then within a finite time (typically, the *stabilization time*), $\mathcal{A}$ will start behaving correctly. However, during the stabilization period, behavior of $\mathcal{A}$ is unpredictable, *i.e.*, several requested computations may violate the intended specification. If $\mathcal{A}$ is snap-stabilizing, then upon a request, it starts within finite time and its behavior after the starting action will be as per its specification. This difference shows the extra power of snap-stabilization with respect to self-stabilization: snap-stabilization system provides stronger safety properties.

Normal and abnormal configurations. Since, regardless the initial configuration, every execution of a snap-stabilizing algorithm satisfies the intended specification, every configuration is legitimate and so legitimacy is no longer relevant in this context. In contrast, we characterize a configuration using the notions of *normal* and *abnormal* configurations (defined below), which are specific to the snap-stabilizing paradigm.

In (self- or snap-) stabilizing systems, we consider the system immediately after transient faults cease. That is, we study the system starting from a configuration reached due to the occurrence of transient faults, but from which no fault will ever occur. Due to the effect of the faults, this configuration is arbitrary. Now, this configuration is referred to as an (arbitrary) *initial configuration* of the system because it is the initial point of observation in the proofs.

The notion of "initial configuration" used here should be clearly distinguished from the classical notion of initial configuration used in the non fault-tolerant algorithms. To avoid any confusion, this latter notion will be called *normal initial configuration* in the following.

Consider a non fault-tolerant distributed algorithm $\mathcal{A}$ realizing specification $\mathcal{SP}_{\mathcal{P}}$. As explained before, the correctness of such an algorithm is established by observing the system from a pre-defined configuration from which the system is supposed to start, the *normal initial configuration*. In this configuration, initiators are enabled for executing a starting action, and the other processes are quiescent, *i.e.*, they are disabled until being involved in a computation initiated by some initiator.

Of course, the concept of normal initial configurations also makes sense in (self- and snap-) stabilizing systems. For example, in self-stabilizing token circulation algorithms, the normal initial configuration is usually the configuration where all processes are in the state "idle." Besides, in case of a real deployment, the network should be initialized in the normal initial configuration of the stabilizing algorithm.

Any configuration reachable from a *normal initial configuration* will be called a *normal configuration*. Any execution starting from a normal initial configuration is called a *normal execution*. Considering faulty networks, the system may be in a configuration which is unreachable from a normal initial configuration. We

call this type of configurations *abnormal configurations*. Using available local information, some processes, so-called *abnormal processes*, may detect that the configuration is abnormal: nothing but a transient fault can explain such an inconsistency. In contrast, some processes, henceforth called *normal processes*, may not detect any inconsistency, *e.g.*, in the atomic-state model, the state of a normal process is consistent with those of its neighbors. Hence, in normal configuration, no process is abnormal, and so all processes are normal.

Delay and zero stabilization time. Again, since, regardless the initial configuration, every execution of a snap-stabilizing algorithm satisfies the intended specification, every configuration is legitimate and, consequently, the stabilization time of a snap-stabilizing algorithm is 0. This assertion is often misunderstood, even in the self-stabilizing community. This is probably due to a confusion between the notions of *delay* and *stabilization time*. Actually, from an arbitrary configuration, the *delay* is the maximum time between a request and the subsequent execution of a starting action. So, the delay of a snap-stabilizing algorithm is nearly always not null.

The notion of delay can be understood using the following analogy. Consider an algorithm that manages a printer. Assume one requests to print a file, and then makes the same request for a second file. So the printing of the second file is delayed until the previous printing has finished. Take the second request as the initial point of the observation: we observe a delay before the printing algorithm starts for this request. So, the notion of delay exists beyond the stabilizing systems. Moreover, the notion of delay and stabilization time are clearly different. The impact of the delay is to slow down the algorithm, whereas during the stabilization time, the specification of the algorithm is violated.

A matter of perspective. Beyond the stronger safety guarantees, maybe the main contribution of snap-stabilization is to think a behavior through actions rather than states: correct a configuration is not a primary target, but rather a mean to obtain a correct result from a distributed computation. For example, assume that your problem is to detect whether there exists a process having an input x equal to 10. A self-stabilizing designer will probably design a proactive algorithm that first corrects the local states of all processes so that it becomes easy to deduce the correct answer. In contrast, the snap-stabilizing designer will design a search process that will backtrack as soon as it meets a process having its input x equal to 10. In many cases, when the (reactive) search succeeds, the computation will not involve all processes, leaving the system in a still quite weird configuration. Notice that this idea is similar to that of Arora and Nesterenko [AN05] to unify stabilization and termination in message-passing systems.

1.6 Efficiency in Self-stabilization

I envision *efficiency* of self-stabilizing algorithms in a broad sense, *i.e.*, it should not only include complexity measures, but also the benefits of the solution both in terms of fault-tolerant property and quality of the computed result. Below, I give and justify a non-exhaustive list of features that I relate to my understanding of efficiency.

1.6.1 Time Complexity

1.6.1.1 Problem-specific Measures

Of course, stabilization time definitely remains the main time complexity measure to compare self-stabilizing algorithms. However, performances of self-stabilizing algorithms must be also evaluated at the light of

problem-specific measures, *e.g.*, the *waiting time*[16] for resource allocation problems (*n.b.*, further measures devoted to resource allocation problems will be studied in Chapter 3), the *cost of a wave* for wave algorithms (see [Cou09b] for more complexity metrics about wave algorithms). Usually, those complexities are evaluated starting from a legitimate configuration, following the principle of separation of concerns. In addition, this allows to evaluate the *time overhead* of the self-stabilizing solution, which is evaluated as the ratio between a time complexity of the self-stabilizing algorithm (starting from a legitimate configuration) and the corresponding time complexity of the best known non-self-stabilizing algorithm for the same task.

As explained before, for many problems, the time overhead of self-stabilizing algorithms is negligible, illustrating then the lightweight nature of self-stabilization. For example, most of the self-stabilizing token passing algorithms, *e.g.*, the snap-stabilizing solutions we have proposed in [CDPV06, CDV09a], achieve each full traversal of the network in $O(n)$ rounds, like non-self-stabilizing solutions (for which a lower bound in $\Omega(n)$ has been proven), giving then a time overhead in $O(1)$.

1.6.1.2 Time Complexity Analysis

Usually self-stabilizing algorithms are analyzed by proving time upper bounds. However, it also important to exhibit worst-case scenarios (as we have done for our leader election algorithm [ACD+17b], for example), at least to show whether the proven upper bound is tight.

Furthermore, by definition, the stabilization time is impacted by worst case scenarios which are often unlikely in practice. So, in many cases, the average-case time complexity may be a more accurate measure of performance assuming a probabilistic model. However, the arbitrary initialization, the asynchronism, the maybe arbitrary network topology, and the algorithm design itself often make the probabilistic analysis intractable. In contrast, another approach consists in *empirically* evaluating the average-case time complexity via simulations. Despite this latter approach is still unusual in the self-stabilizing area, I think that every formal time complexity analysis should be always coupled with experimental results. Indeed, the average behavior of an algorithm is often notably better than its upper bound (once again because the worst-case is often unlikely). As illustrative example, the simulations of our leader election algorithm given in [ACD+17b] show a round complexity that is linear on the diameter of the network on average while we have established a stabilization time in $\Theta(n)$ rounds, where n is the number of processes.

Another interesting approach, for resource allocation problems for example, can be the *amortized analysis*. This latter is also quite unusual in the self-stabilizing area, however such an analysis may be interesting in some cases, *e.g*, in [CDV09b] Cournier *et al.* propose a routing algorithm for message-switched networks which achieves, in the worst case, the routing of a single message in a time exponential in Δ (the maximum degree of the network), yet since the worst-case is very unlikely, they could also show that the amortized cost of message routing is linear.

Another important line of research is to study important classes of specifications and give for them significant time lower bounds, *e.g.*, Genolini *et al.* [GT02] considered algorithms that are self-stabilizing for particular non-static specifications and exhibited a lower bound of $\Omega(\mathcal{D})$ rounds on their stabilization time, where $\mathcal{D}$ is the diameter of the network. Coupled with tight upper bounds on algorithms, such lower bounds allow to establish time optimality of self-stabilizing solutions. Time optimal self-stabilizing solutions have been proposed for various problems, *e.g.*, token circulation [PV99], clock synchronization [AKM+93], and spanning tree constructions [KK13], to only quote a few.

[16] *I.e.*, the maximum time for a requesting process to enter the critical section.

1.6.2 Space Complexity

1.6.2.1 Space Complexity Measures

The space complexity of algorithms is usually evaluated in terms of *number of (local) states* or *memory requirement*. The memory requirement of a process is the number of bits it requires to store all its variables, *i.e.*, the sum of the ceiling of the logarithm to the base 2 of the domain size of its variables. Hence, overall, the memory requirement of a process is logarithmic in the number of its local states.

Another important metric is the *overhead in space* of a self-stabilizing algorithm. It consists in studying the ratio between its space complexity and the space complexity of the best known non-self-stabilizing algorithm for the same task. Notice that there are problems, such as PIF in rooted (directed) tree, where self-stabilizing solutions with a space overhead in $O(1)$ have been proposed [BDPV99a], justifying again the lightweight nature of self-stabilization.

1.6.2.2 Efficiency in Space

Designing efficient self-stabilizing algorithms in terms of space complexity is rather challenging. Several papers investigate space optimality of self-stabilizing algorithms [Her92a, Joh97, BDPV99a].

However, there are problems where space and time complexities are orthogonal issues: there are self-stabilizing algorithms that are optimal w.r.t. each of them, but not for both. For example, in [BDPV99b], authors show that a self-stabilizing PIF algorithm in undirected tree cannot be both optimal in space and time, while it can be done separately. In [BDGPB$^+$10], we have proposed a probabilistically self-stabilizing vertex-coloring algorithm that uses k states per process, where k is a parameter of the algorithm. When $k = \Psi + 1$ where Ψ is the maximum number of processes that are linked (using an outgoing or incoming link or both) to a process of the network, the expected stabilization time is $O(\Psi \cdot n)$ moves, where n is the number of processes. When k grows arbitrarily, the algorithm recovers in $O(n)$ expected moves in total. So, this solution can be tuned to be either optimal in space (and is then with a Ψ multiplicative penalty in time), or optimal in time, but not both.

Conversely, there exist few algorithms that are optimal both in space and time, *e.g.*, the two snap-stabilizing (depth-first) token circulations respectively designed for oriented and rooted trees proposed in [PV99].

Finally notice that, in abstract model such as the atomic-state model, space complexity is of particular interest since variables are locally shared, meaning that variables are continuously read by all neighbors. So, this gives the communication footprint of the algorithm. Besides, the translation of an algorithm from the atomic-state model to the message passing model usually requires the use of heartbeats, where each process regularly transmits all or part of its local state to all its neighbors; see [ADDP19a].

1.6.3 Volume of Exchanged Data and Communication-efficiency

The volume of exchanged data, also called *bit complexity*, evaluates the number of information bits exchanged between neighboring processes. Due to the impossibility of local detection of stabilization (*cf.*, Section 1.3.7), this amount is usually unboundable in the self-stabilizing context. Indeed, except in the work of Arora and Nesterenko [AN05], most of existing self-stabilizing algorithms are proactive, with the shortcoming that once the system is stabilized, processes should regularly exchange control data. Now, since, by definition, transient faults are infrequent, if we are able to propose a self-stabilizing algorithm with a small stabilization time, then the system behaves correctly most of the time and so those control data are most of the time useless.

To answer this issue, we have imported the concept of *communication-efficiency* in self-stabilization [DMT09, DDF10]; this latter was previously used in the context of crash-prone systems [LFA00]. The main idea behind this concept is to try to reduce the number of links through which information are sent infinitely often.

This can be done globally, as we propose in [DDF10] a self-stabilizing leader election algorithm for synchronous fully-connected message passing systems that uses only $n-1$ of the $O(n^2)$ available links (n being the number of processes) once the system has stabilized. Actually, using these links, the elected process regularly sends alive messages to all other processes in order to be no more suspected.

In [DMT09], we have proposed a local version of communication-efficiency. For example, we provide, in the atomic-state model (assuming a distributed unfair daemon), a silent self-stabilizing algorithm that computes a dominating (independent) set. Once this latter has stabilized, the processes of the dominating set, called *dominators*, continue to regularly read the state of all their neighbors, while all other processes only check the local state of one neighbor, their dominator.

Notice that communication-efficiency is particularly well-suited for silent tasks and have led to many other works; see, *e.g.*, [TOKM12, DLM14].

1.6.4 Strong Forms of Self-stabilization and their Specific Metrics

As explained before, the quality of a self-stabilizing solution should also be assessed at the light of the fault tolerance property it guarantees. Thus, strong forms of self-stabilization have been introduced to capture those additional skills.

In particular, some of them are defined as parameterized property to precisely catch those skills. For example, in [ADDP19b] we have introduced a specialization of self-stabilization called *gradual stabilization under* (τ,ρ)*-dynamics*. An algorithm is gradually stabilizing under (τ,ρ)-dynamics if it is self-stabilizing and satisfies the following additional feature. After up to τ *dynamic steps* of type ρ occur starting from a legitimate configuration, a gradually stabilizing algorithm first quickly recovers a configuration from which a specification offering a minimum quality of service is satisfied. It then gradually converges to specifications offering stronger and stronger safety guarantees until reaching a configuration (1) from which its initial (strong) specification is satisfied again, and (2) where it is ready to achieve gradual convergence again in case of up to τ new dynamic steps of type ρ. Such parameters allow to quantify the strength of those additional skills, and study both lower bounds and optimality w.r.t. them. (Further detail about gradual stabilization will be given in Chapter 4.)

Then, several metrics devoted to strong forms of self-stabilization have been introduced to precisely analyze algorithms w.r.t. their additional features. Below we give some illustrative examples.

In snap-stabilizing systems, we almost only consider dynamic specification. So, the *delay*, which is the maximum time between a request and the subsequent execution of a starting action, is of prime interest.

In superstabilization, two additional complexity measures are meaningful: the *superstabilizing time* and the *adjustment measure*. The superstabilizing time is the maximum time it takes for a superstabilizing algorithm to recover after a single topological change occurs in a legitimate configuration. The adjustment measure is the maximum number of nodes that have to change their state during this later recovery process.

Similarly to superstabilization, in fault-containment, supplementary measures evaluate the system from a configuration γ reached after few faults, say less than k, occur in a legitimate configuration. The *containment time* then evaluates the maximum time necessary for the algorithm to recover from γ. The *contamination number* evaluates the maximum number of processes that will be involved in this recovery process. Finally, the *fault gap* is the maximum time from γ before the system is ready again to efficiently handle less than k faults.

1.6.5 Quality of the Computed Solution

In many problems such as the construction of distributed structures (*e.g.*, a spanning tree, a dominating set), the quality of computed solution is an important issue and so should be evaluated. For many such problems, optimality is too expensive. For example, finding a k-clustering with the minimum number of k-clusters is $\mathcal{NP}$-hard [GJ79]. On the other hand, efficiently computing a solution that does not offer strong guarantees on the quality of the result may lead to degenerated solutions that are not interesting in practice. For example, a trivial solution to the dominating set problem consists in putting all the nodes in the set.

Hence to achieve a good trade-off, many works propose time and space efficient solutions that offer additional guarantees related to the quality of the computed solution, *e.g.*, quantitative bounds, approximation ratio, [17] or inclusion-wise maximality or minimality.

For example, in the k-clustering problem, which consists in partitioning processes into clusters of radius at most k, the aim is to minimize the number of clusters and so we can expect a bound on the number of built clusters and even a proof that the solution achieves a good approximation ratio. In [DDL19], we have proposed a silent self-stabilizing algorithm that builds a k-clustering of the network made of at most $O(\frac{n}{k})$ clusters. In [DDH$^+$16], we have proposed a silent self-stabilizing k-clustering algorithm that builds at most $O(\frac{n}{k})$ clusters in the general case and additionally ensures a $O(k)$ approximation ratio when the network topology is actually a unit disk graph.

We have presented silent self-stabilizing algorithms that compute inclusion-wise minimal solutions for various problems, including dominating sets [DMT09], generalized k-dominating set problems [DDL19], and (f,g)-alliances [CDD$^+$15]. By inclusion-wise minimality, we mean that we compute a set of processes that satisfies the definition of the problem while none of its proper subsets does.

Finally, we have given a silent self-stabilizing 3-approximation algorithm for the maximum leaf spanning tree problem in arbitrary networks [DKKT10].

1.6.6 Simplicity, Assumptions, and Models

Simplicity should be a goal in algorithmics, not only for the beauty of art, but rather to (at least) increase the confidence on the correctness of a solution.

Indeed, in distributed computing (in particular in self-stabilization), both correct operation and complexity of distributed algorithms are commonly established using paper-and-pencil proofs. This potentially leads to errors when arguments are not perfectly clear, as explained by Lamport in its position paper [Lam12]. To prevent flaws in proofs, simplicity is a valuable answer. Moreover, simplicity makes easier the mechanical checking of the proof using a *proof assistant*. For example, one can use the general framework based on the proof assistant Coq[18] we have developed [ACD17a] to certify the self-stabilization of a given algorithm.

Simplicity also goes hand in hand with limiting the assumptions in the modeling. A huge number of narrow assumptions may lead to an inconsistent modeling since a combination of them may be contradictory. Limiting assumptions is also meaningful in terms of flexibility and practicality. For example, reducing the required initial knowledge on the network (such as an upper bound on the diameter or the number of processes, for example) or the level of distinguishability (*i.e.*, fully anonymous, semi anonymous, or identified) makes the algorithm easier to deploy and increase its application domains. Avoiding fairness and synchrony assumptions makes the design more realistic. In the same spirit, the quality of the solution should be judged at the light of the computational model it assumes.

[17] An approximation ratio α means that the solution is at most α times bigger than the optimal.

[18] `https://coq.inria.fr/`

1.7 Versatility in Self-stabilization

By *versatility* I both consider expressiveness in a broad sense, general algorithmic constructions, and general complexity bounds.

1.7.1 Expressiveness

By expressiveness, I mean exploring the limit of a given property, here related to self-stabilization. For example, Katz and Perry [KP93] have addressed the expressiveness of self-stabilization in message passing systems where links are reliable and have unbounded capacity, and processes are both identified and equipped of infinite local memories. Precisely, they have characterized which specifications admit a self-stabilizing solution under such settings. Of course, such a study should be done for each variant of self-stabilization. As an illustrative example, Chapter 2 will be dedicated to the work we have done on the expressiveness of snap-stabilization in the atomic-state model [CDD$^+$16].

Expressiveness also deals with computational models, *i.e.*, are two models equivalent in terms of computational power? Equivalence (in terms of computational power) between the atomic-state model and the register one and between the register model and message passing are discussed in [Dol00, ADDP19a]. Assumptions also affect expressiveness, *e.g.*, many problems have no self-stabilizing solutions when the processes are assumed to be anonymous [BDGPB$^+$10, DTY15]. Such impossibility results are in particular important since they motivate the introduction of weak versions of self-stabilization. For example, leader election in trees or token rings have no self-stabilizing solution in a fully anonymous system; see [YK96] and [Her90], respectively. Now, in [DTY15], we give weak stabilizing solutions for these two problems. In the same spirit, computing optimized distributed structure self-stabilizingly may be very costly. In such cases, computing a tight approximation makes sense; see *e.g.* our work on the maximum leaf spanning tree problem in arbitrary networks [DKKT10].

In a broad sense, expressiveness also includes complexity issues, *i.e.*, when a problem or a class of problems is solvable, a natural question arise: "at which cost?". As explained before, space and time lower bounds should be established, as much as possible, for the largest possible specification classes.

1.7.2 General Algorithmic Schemes

Many different general algorithmic schemes have been used along the years in the self-stabilizing area. Below, I give a non-exhaustive list of techniques we have used to implement general self-stabilizing constructions.

1.7.2.1 Transformers

A popular generic approach in self-stabilization is to propose *transformers* or *compilers*, *i.e.*, meta-algorithms that transform an input algorithm, which has not a desired property (*e.g.*, a non-self-stabilizing algorithm), into an algorithm which achieves that property.

Such transformers are useful to establish expressiveness of a given property: by giving a general construction, they allow to exhibit a class of specifications that can achieve a given property. An impossibility proof should be then established to show that the property is not achievable out of the class, giving then a full characterization. For example, in Chapter 2, I will present our general transformer which transforms an algorithm into a snap-stabilizing one for the same specification. Notice that transformers have been also used to emulate abstract models into more practical ones; see, *e.g.*, [Dol00, ADDP19a].

It is important to note that versatility is often obtained at the price of inefficiency: transformers often use heavy mechanisms such as global snapshots and resets in order to be very generic. However, transformers can

be also used to show that a class of specifications can be efficiently treated. To be more precise, transformers may be used to show that an efficient algorithmic concept may be widely employed. For example, we show in [CDV09a] that, in the atomic-state model, non-self-stabilizing wave algorithms yet implemented with an additional very light property (related to termination) can be turned into efficient snap-stabilizing algorithms. In the same spirit, we have shown how to make efficiently snap-stabilizing self-stabilizing wave algorithms [CDV06a], still in the atomic-state model. Finally, in [DJ19] we propose a reset-based transformer to give time-efficient (both in terms of rounds and steps) self-stabilizing versions of particular non self-stabilizing algorithms written in the atomic-state model.

1.7.2.2 Composition Techniques

Composition techniques are widely used to design self-stabilizing algorithms [Tel01]; see, *e.g.*, [KC99, BFP14, FYHY14, DDH$^+$16, ADD17b]. A composition is usually modeled using a binary operator, say $\circ$, over two distributed *sub-algorithms* $\mathtt{A}$ and $\mathtt{B}$. This operator defines how the two sub-algorithms run together. The resulting algorithm, noted $\mathtt{B} \circ \mathtt{A}$, is called a *composite algorithm*. Composition operators are provided with sufficient conditions to show the self-stabilization of a composite algorithm. Hence, their main purpose is to simplify both the design and the proofs of self-stabilizing algorithms, following a divide and conquer approach.

Notice that a composite algorithm is meant to be an algorithm defined in the considered model. So, the composition operator should be carefully formalized as it should not increase the expressive power of the model. That is, it should not assume properties that cannot be implemented in the model. In other words, there must exist a rewriting method to translate the composite algorithm into an algorithm that achieves the same specification, yet without using the composition operator.

Various composition techniques have been introduced so far, *e.g.*, *collateral composition* [GH91, Her92b], *fair composition* [Dol00], *cross-over composition* [BGJ01], and *conditional composition* [DGPV01]. Actually, each of those aforementioned composition techniques allows to implement one of the two following algorithmic design patterns.

1. The former approach involves first the design of a self-stabilizing algorithm $\mathtt{A}$ that emulates a strong scheduler (*e.g.*, a sequential one). Then, $\mathtt{A}$ is used to control the execution of another algorithm $\mathtt{B}$, limiting then the number of possible interleavings among actions of $\mathtt{B}$. Hence, $\mathtt{B}$ can be designed and proven under such strong scheduling assumptions.

   For example, in the atomic-state model, a self-stabilizing token circulation can be used to emulate a central daemon under a distributed daemon assumption. Then, one can design a self-stabilizing algorithm for a desired specification assuming a central daemon.

   The composition then makes the first algorithm controlling the execution of the second one so that it respects the specification of the strong daemon. Consequently, the composite algorithm self-stabilizes to the expected specification under the more general daemon (in our example, a distributed daemon).

   To enforce sequentiality in our example, the composition enables a process to execute an action of the second algorithm only right before releasing the token.

2. The latter approach consists of first self-stabilizingly solving the desired specification on a restricted class of networks, say $\mathcal{N}'$ (*e.g.*, trees or rings), and then bringing the solution to a more general class of topologies, say $\mathcal{N}$ with $\mathcal{N}' \subsetneq \mathcal{N}$ (usually arbitrary connected networks).

   To that goal, we implement a self-stabilizing algorithm that builds a virtual structure of type $\mathcal{N}'$ (*e.g.*, a spanning tree) on the top of a network of type $\mathcal{N}$. Then, in the composition, the algorithm for

topologies $\mathcal{N}'$ runs on a network of type $\mathcal{N}$, but only considering the links belonging to the virtual structure, which is eventually of type $\mathcal{N}'$.

The first approach can be applied using the *cross-over composition* [BGJ01], for example. In [DLD+13], we have introduced the *hierarchical collateral composition* to cope with the second approach. This composition technique is actually a straightforward variant of the *collateral composition* introduced by Herman [Her92b]. In the collateral composition, the composition of two algorithms just consists of running the two algorithms concurrently. The second algorithm usually takes the output of the first one as input. For example, the first algorithm can be a silent spanning tree construction and the second one can be an algorithm dedicated for tree topologies that takes the output parent pointers of the first one as input. Hence, once the first algorithm has converged, the second one is in an arbitrary configuration, except that the parent pointers are constant and describe a spanning tree. From that point, the stabilizing property of the second algorithm ensures the overall stabilization of the composite algorithm.

The hierarchical variant of the collateral composition is justified by the following problem. When two actions are enabled at the same process but in two different algorithms of the composition, the process *nondeterministically* executes one or the other, if activated by the daemon. The hierarchical collateral composition solves this nondeterminism using local priorities: in the hierarchical collateral composition of `A` and `B`, the code of the local algorithm $\texttt{B}(p)$ (for every process p) is modified so that p executes an enabled action of $\texttt{B}(p)$ only when it has no enabled action in $\texttt{A}(p)$. Thus, locally at each process, actions of `A` have priority over actions of `B`. However, we should underline that the priorities are only local: an enabled action of `B` can be executed by some process p (in particular because p has no enabled action in `A`), while other processes in the network have enabled actions in `A`.

In [DLD+13], we have defined the *hierarchical collateral composition* and proposed a very simple sufficient condition to establish the self-stabilization of a composite algorithm $\texttt{B} \circ \texttt{A}$ to the specifications of both `A` and `B` under a weakly fair daemon: namely, stabilization is guaranteed whenever `A` is silent and `B` stabilizes starting from any configuration where no action in `A` is enabled. The simplicity of this condition allowed us to use it in order to solve various problems, *e.g.*, k-clustering [DLD+13, DDH+16], ℓ-exclusion [CDDL15], and ranking [DLDR13]. Moreover, again thanks to its simplicity, we were able to certify (*i.e.*, mechanically check) the proof of the sufficient condition using our framework based on the proof assistant Coq; see [ACD19].

1.7.2.3 Parametric Solutions

Another way to design general solutions consists in considering parametric specifications. For example, ℓ-exclusion problem [FLBB79] is a generalization of mutual exclusion, where up to $\ell \geq 1$ critical sections can be executed concurrently, instead of at most one.

There are many ways to envision a parametric specification. Parameters can be just integers, like in my works on k-clustering [DLD+13, DDH+16], ℓ-exclusion [CDDL15], and k-out-ℓ-exclusion [DDHL11].

Yet, parameters may be of higher order. For example, in [CDD+15] we have considered the problem of constructing a (f,g)-alliance, where f and g are two non-negative integer-valued functions on processes. The (f,g)-alliance can be instantiated into several classical constructions, including the dominating set problem (*i.e.*, a $(1,0)$-alliance).

In [ADD17b], we have studied a generalization of local resource allocation problems called *LRA*, where the safety of the problem is defined thanks to a relation called *compatibility relation*. This relation allows to define several local resource allocation problems, such as the local mutual exclusion, the local readers-writers problem, and the local group mutual exclusion. Further details will be given in Chapter 3.

In [DDL19], we have proposed a generalization of the minimal k-dominating set problem: given a positive integer k and two sets of processes $R^* \subseteq D^*$, the problem is to find a set D which is minimal subject to the conditions that $R^* \subseteq D \subseteq D^*$ and that D is k-dominating relative to D^*, meaning that every process within distance k of D^* is also within distance k of D. The original minimal k-dominating set problem is then the special case that $D^* = V$ and $R^* = \emptyset$, where V is the set of processes.

Finally, in [DIJ19] we propose a general scheme to compute spanning-tree-like data structures on bidirectional weighted networks of arbitrary topology. This scheme can be instantiated to efficiently solve several spanning-tree-like data structures such as shortest-path, depth-first search (DFS), and breadth-first search (BFS) spanning trees. Further details will be given in Chapter 5.

1.7.2.4 Top-down and Bottom-up Approaches

Following a reverse engineering approach, we can analyze existing algorithms, find similarities, and exploit them in order to formalize a general algorithm class together with associated general (tight) bounds.

For example, many silent self-stabilizing algorithms dedicated to tree topologies or deployed in a network where a spanning tree or forest is available use *top-down* and *bottom-up* approaches [KC99, Dev05, DDH+16].

In [ADD18], we have formalized a wide class of algorithms written in the atomic-state model which use such patterns. Interestingly, this class is defined by a simple (*i.e.*, quasi-syntactic) condition, making the membership test very easy. We have shown that all algorithms of this class are silent and self-stabilizing under the distributed unfair daemon. Moreover, we have exhibited tight bounds in the stabilization time of these algorithms: the stabilization time is both polynomial in moves and asymptotically optimal in rounds.

1.7.2.5 Versatility versus Practicability

The generality of a meta-algorithm often lead to a too complex solution with too many high-order parameters. This makes the scheme difficult to be instantiated. In the same spirit, it may make the sufficient condition or the complexity analysis intractable. Hence, a trade-off should be achieved between generality and usefulness of a general scheme.

1.8 Roadmap

The remainder of this report will be devoted to few contributions I believe to be important and that underline the two axes I want to investigate here, *i.e.*, efficiency and versatility in a broad sense in the context of self-stabilization. In Chapter 2, we study the expressiveness of snap-stabilization in the atomic-state model under the distributed unfair daemon. Precisely, we show that, in this model, self- and snap-stabilization have the same expressiveness in identified connected networks. In Chapter 3, we address the problem of tightly capturing the concurrency properties of (self-stabilizing) resource allocation algorithms. In Chapter 4, we present three derived forms of self-stabilization we have introduced. We discuss the advantages and drawbacks of these properties. Moreover, we compare them to existing derived forms of self-stabilization. In Chapter 5, we present methods to unify both versatility and efficiency in self-stabilizing distributed systems. Finally, in Chapter 6, we present some perspectives, following three main axes: handling more uncertain environments, more accurately capture efficiency, and targeting more general solutions.

Chapter 2

Expressiveness of Snap-stabilization

This chapter is a summary of a paper we have published in the international journal Theoretical Computer Science; please refer [CDD+16] for more details.

2.1 Flashback on the Expressiveness of Self-stabilization

In 1993, Katz and Perry [KP93] considered the expressiveness of self-stabilization in message passing systems where links are reliable and have unbounded capacity, and processes are both identified and equipped of infinite local memories. In this model, they propose a characterization of specifications admitting a self-stabilizing solution. Namely, they show that a non-stabilizing algorithm can be transformed into a self-stabilizing algorithm for the same specification if and only if its specification satisfies a given property. Actually, they have not named this property. Let us call it the *Kleene-closed* property by analogy with the Kleene closure from the language theory. Hence, the result of Katz and Perry can be formulated as follows.

A non-stabilizing algorithm can be transformed into a self-stabilizing algorithm for the same specification if and only if its specification is Kleene-closed.

Roughly speaking, a specification is *Kleene-closed* if it excludes any infinite execution which is decomposable into a finite prefix p and a suffix s where a particular non-empty behavior occurs in p, but never in s.

A trivial non Kleene-closed specification is the problem of "printing" the infinite sequence 1, 0, 0, 0 ... Such a problem trivially admits no self-stabilizing solution. Indeed, assume an algorithm `A` that realizes this specification in an execution e (*n.b.*, any self-stabilizing algorithm realizes its specification in at least one execution, by the correctness property). By definition, e has an infinite suffix s in which no 1 is ever printed. Then, s is a possible execution, which starts from an arbitrary configuration, and s has no suffix satisfying the specification: consequently `A` is not self-stabilizing since it does not satisfy the convergence property.

It is worth noticing that, even if Katz and Perry consider a message passing model, the impossibility side of their characterization, *i.e.*, the fact that non Kleene-closed specifications have no self-stabilizing solution, is independent from the computational model. Indeed, essentially the impossibility holds because processes cannot be confident with their local state due to the arbitrary initialization of the system. In the example, processes cannot decide to never output 1 anymore because they have no means to remember it has been already done (*e.g.*, they cannot trust their local memory since this latter may be arbitrarily corrupted). As a matter of fact, this impossibility result holds even considering a central system, *i.e.*, a single process; in this case, the process can access to the global state of the system directly.

2.2 Contribution

Our goal is to compare the expressive power of self- and snap- stabilization. To that goal, we consider the atomic-state model under the distributed unfair daemon. Moreover, we assume that problems are defined in terms of *dynamic specification*.[1] In this context, we show that, for every identified connected network, self- and snap- stabilization have the same expressiveness, meaning that for any problem for which there exists a self-stabilizing solution, there also exists a snap-stabilizing one, and *vice versa*. By definition, snap-stabilizing algorithms are self-stabilizing algorithms whose stabilization time is null. So, the only interesting question is to demonstrate the first part of our assertion.

To answer this question, we have first designed a snap-stabilizing *Propagation of Information with Feedback* (PIF) for rooted networks of arbitrary connected topology.[2] This algorithm is proven assuming the distributed unfair daemon, the most general daemon.

Then, we use this snap-stabilizing PIF to implement snap-stabilizing versions of four other fundamental distributed algorithms: *Reset*, *Snapshot*, (Guaranteed Service) *Leader Election*, and *Termination Detection*.

Based on all these algorithms, we design, still assuming a distributed unfair daemon, a *universal transformer* that provides a snap-stabilizing version of any algorithm which can be self-stabilized using the transformer described by Katz and Perry [KP93].

Note that our purpose is only to demonstrate the feasibility of transforming almost any algorithm (specifically, those algorithms that can be self-stabilized) to a corresponding snap-stabilizing algorithm. As a consequence, our method is inefficient due to its versatility. Recall that the trade-off between versatility and efficiency will be discussed in Chapter 5.

2.3 Propagation of Information with Feedback (PIF)

2.3.1 Definition

The concept of *Propagation of Information with Feedback* (PIF), also called *Wave Propagation*, has been introduced by Chang [Cha82] and Segall [Seg83]. PIF has been extensively studied in the distributed literature because many fundamental problems, *e.g.*, *Reset*, *Snapshot*, *Termination Detection*, *etc.* can be solved using a PIF-based approach.

A PIF algorithm is a *wave algorithm* [Tel01], *i.e.*, each of its execution can be partitioned into fragments called *waves* such that each wave w satisfies the following requirements:

1. w is finite,
2. w contains at least one special event called *decision*, and
3. each decision event at some process in w is causally preceded (in the sense of Lamport [Lam78]) by at least one event in w at each other process.

Specifically, a PIF-wave must at least contain the following two phases:

- a broadcast phase where a data should be propagated in all the network and delivered (to the application layer) once by each process, and
- a feedback phase where processes acknowledge the receipt of the data to the initiator of the broadcast.

[1] Recall that dynamic specifications are defined and their role is justified starting from page 23.

[2] This algorithm has been initially presented at ICPADS'2006 [CDV06b].

After the feedback phase, the initiator decides (the only decision event) and a new broadcast can be initiated and so on.

More precisely, a process, called *initiator*, starts the first phase of the PIF (*i.e.*, the broadcast phase) by initiating the broadcast of some data d to all processes of the network.

Then, each non-initiator acknowledges the receipt of d to the initiator during the *feedback phase*. The feedback phase is usually executed bottom-up into the tree defined by the broadcast phase: the initiator is the root of the tree and each other process chooses as parent in the tree the process from which it receives the data d first. In more details, the leaves initiate the feedback phase by sending an acknowledgment to their parent when they detect that d has been received by all their neighbors. Then, a non-root internal node sends an acknowledgment to its parent (1) after receiving an acknowledgment from each of its children and (2) providing that all its neighbors have participated to the PIF-wave. The feedback phase terminates at the initiator when (1) it has received an acknowledgments from all its children and (2) all its neighbors have participated to the PIF-wave. Hence, at the end of the feedback phase the decision event of the initiator is causally preceded by the acknowledgment of the reception of d by all other processes.

Notice that, in arbitrary distributed systems, any process may need to initiate a PIF-wave. Thus, any process can be the initiator of a PIF-wave and so several instances of PIF algorithms may run simultaneously. To cope with the concurrent executions, every process maintains the identity of the initiators. However, we first study here the PIF problem in a general setting where we consider only one instance of PIF algorithm. For this instance, the initiator will be called the *root* and denoted by r.

Recall that to be snap-stabilizing, a PIF algorithm should satisfy the following two conditions in each of its execution: (i) if r has a data d to broadcast, it will initiate that broadcast within a finite time, and (ii) starting from any configuration where r starts to broadcast d, the system behaves as per its specification: every other process receives d and acknowledges this receipt to r within finite time.

2.3.2 Features of our PIF Solution

The major advantage of our snap-stabilizing solution is that it is proven under the distributed unfair daemon. To the best of our knowledge, this is until now the only snap-stabilizing PIF algorithm for arbitrary connected rooted networks which has been proven in the atomic-state model assuming the distributed unfair daemon. It is also important to note that we obtain this result without degrading the performances. As a matter of facts, the round and space complexities of our solution, resp. $O(n)$ rounds (n is the number of processes) and $O(\log n)$ bits per process,[3] match the previous results [CDPV02, BCV03]. However, contrary to the previous solutions, we could exhibit a bound on its move (and step) complexities, namely $O(\Delta \times n^3)$ moves (Δ is the maximum degree of the network), as it works under an unfair daemon. Notice that, our solution is still today the best existing for rooted networks of arbitrary connected topology.

2.3.3 Overview of our PIF Solution

We now present the main principles of our solution, called Algorithm $\mathcal{PIF}$ in the following. A wave of Algorithm $\mathcal{PIF}$ is split into the following three consecutive phases: the broadcast, feedback, and cleaning phases. To implement these three phases, each process has three variables: a status variable, a parent pointer, and a distance variable (the two latter are constants for r, respectively equal to $\perp$ and 0). Either the process has status $idle$, meaning that it is currently not participating in a wave, or its status indicates in which phase it is involved (say B for broadcast, F for feedback, and C for cleaning). The parent pointer is used to designate

[3] Providing that the knowledge of an upper bound on n which is in $O(n)$ is available.

the neighbor from which the process receives the data first in the broadcast phase. The distance variable gives the level of the process in the tree described by the parent pointers.

2.3.3.1 Normal Execution

A configuration of $\mathcal{PIF}$ is normal initial whenever every process has status $idle$. Starting from a normal initial configuration, the PIF-wave is executed as follows. Upon a request, the root process r initiates the wave by switching to the broadcast phase. Then, a non-root process switches to the broadcast phase when at least one of its neighbors is in the broadcast phase, it also designates its parent and evaluates its distance in the same step. The feedback phase is executed bottom-up, starting from the leaves: a process switches to the feedback phase when it is in the broadcast phase, all its neighbors are involved in the PIF-wave (precisely with status B or F), and all its children (if any) are in the feedback phase. Note that the last condition is trivially satisfied by the leaf processes since they have no children.

Right after r switches to the feedback phase, the system is in a configuration where all processes are involved in the feedback phase. Then, the root initiates the cleaning phase by broadcasting in the tree the status C. When the cleaning status reaches the leaves, the system is reset to the $idle$ status in a bottom-up fashion so that when r switches to the $idle$ status the reached configuration is normal initial again.

2.3.3.2 Error Correction

Consider now the case where the initial configuration of the system is arbitrary. There may be inconsistencies between the states of neighboring processes. To detect inconsistencies, each non-root process compares its state to that of its parent. Any non-root process is said to be *abnormal* if it is involved in a PIF-wave (*i.e.*, its status is not $idle$) and its status or its distance is inconsistent w.r.t. that of its parent (*e.g.*, a process has status B while its parent has status F).

Now, an abnormal process may have children whose states are consistent with its own state, *i.e.*, these children are *normal*. We define an *abnormal tree* rooted at the abnormal process p as the set including p and all its *normal* descendents. (*N.b.*, acyclicity is guaranteed by the fact that a non-root process is abnormal if its distance is not equal to one plus that of its parent.) Conversely, we call *normal tree* the set of process including r and all its *normal* descendents.

All abnormal trees (if any) should be removed so that the system recover a normal configuration. We perform this removal top-down. However, to be efficient we should prevent the following situation: by leaving its tree, an abnormal process p creates some abnormal trees, each of those being rooted at a previous child; and later p joins one of those (created) trees or a tree issued from them.[4] Hence, the idea is to freeze each abnormal tree, before removing it (this mechanism is inspired from [BCV03]). By freezing we mean assigning each member of the tree to a particular state that (1) blocks processes in the tree until they definitely leave it, and (2) forbids new process to join the tree. This freezing mechanism is implemented using two additional error statuses EB and EF (of course, we should also check inconsistencies regarding these two new statuses). These two statuses aim at performing a PIF in the abnormal tree. At the end of that PIF, all members of the tree have an error status (actually EF). So, the tree is frozen and can be simply removed by resetting all its members top-down to the status $idle$.

[4] This issue is close to the *count-to-infinity* problem [LGW04].

2.3.3.3 Question Mechanism

The error correction prevents the system from any deadlock. However, the presence of abnormal trees during the error correction should be carefully addressed to prevent safety violation after r has started the broadcast phase. Precisely, the problem is to prevent any process involved in a correct broadcast phase, *i.e.*, a broadcast phase initiated by r, from switching to the feedback phase too early. Indeed, consider a process p that is involved into a correct broadcast phase and ready to switch to the feedback phase. In this case, p should be sure that all its neighbors are involved in the PIF initiated by r, and so not in an abnormal tree. The goal of the *question mechanism* is to remove such an ambiguity.

The question mechanism is implemented using an additional variable at each process, denoted by *Que*; see Figure 2.1 for an illustration. A process p initiates a question each time it switches to a broadcast phase. The idea is to propagate the question up in the tree of p and in the trees of all non-children neighbors that are involved in a PIF-wave (with status B or F). Then, only r is allowed to emit a (positive) answer. Hence, to be allowed to switch from the broadcast to the feedback phase, a process p should satisfy the following conditions:

- it has received an answer (from r), meaning that it belongs to the normal tree;
- all its children are in the feedback phase; and
- all its non-children neighbors are both involved in a PIF-wave (precisely with status B or F) and have received an answer (from r), meaning that they belong to the normal tree.

Notice that the variable *Que* is used both to transmit the question up and propagate the answer down in trees, and so an additional state is required to remove any ambiguity. Moreover, initially *Que* variables may have arbitrary values. So, we should prevent those values to generate a false answer. To that goal, each time a question is initiated by some process, a reset of the *Que* variables involved in that question is performed bottom-up right before propagating the question using the additional state, in a pipeline manner (in an efficiency concern). A side effect of this reset is that *Que* variables do not need to be considered when checking if a process is abnormal or not. Another important property of the question mechanism is that it allows to eventually block the growing of abnormal trees.

It is worth noticing that this waiting mechanism does not create any deadlock because, thanks to the error correction, eventually all abnormal trees disappear from the system. From that point, a process will be either idle or in the normal tree. In the former case, the process will initiate a question each time it will switch to the broadcast phase and then will receive an answer within finite time, since this broadcast phase will be that of r. For similar reasons, in the latter case, the process will receive an answer from the root within finite time.

Note also we constrain a process can participate to the broadcast phase only if no process involved into a PIF-wave designates it as parent. This simple check together with the question mechanism inductively ensure the correct operation of any PIF-wave initiated by r.

2.4 Other Key Algorithms

Using our snap-stabilizing PIF algorithm we could derive four essential snap-stabilizing algorithms. These latter are very useful to develop our universal transformer.

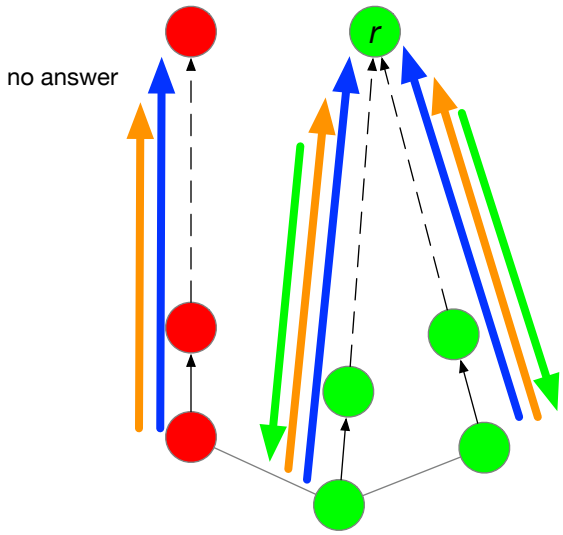


Figure 2.1: Schematic description of the question mechanism. Red nodes belong to an abnormal tree, while green ones belong to the normal tree. Blue, orange, and green arrows show the direction of the reset, question, and answer propagations, respectively.

2.4.1 Snap-Stabilizing Leader Election

Here, we are interested in a weak version of the leader election problem, that we call *guaranteed service leader election* in [AD17]. Each process just checks whether it is the actual leader and only when it needs to. The answers (true or false) should be consistent: for every process p, all requests initiated by p return the same answer; moreover there should exist exactly one process for which the answer to each request it initiates is true. Notice that any snap-stabilizing solution to this problem allows to snap-stabilizingly mimic a rooted network.

To that goal, we first design a snap-stabilizing maximum ID computation using Algorithm $\mathcal{PIF}$, called Algorithm $\mathcal{M}$. Every process in the network runs its own instance of Algorithm $\mathcal{M}$ in parallel. Hence, the guaranteed service leader election algorithm just consists for a requesting process in executing its own instance of $\mathcal{M}$ and then checking whether its identifier is equal to the computed maximum ID. In the following, we refer to this algorithm as Algorithm $\mathcal{LE}$.

2.4.2 Snap-Stabilizing Reset

The *reset* algorithm is used in a faulty environment to "reset" the system to a pre-defined "good" configuration for a problem $\mathcal{P}$. A good global configuration can be restricted to a normal initial configuration to simplify the design process. Yet, any normal configuration can be used as a good configuration. Our approach to design the reset algorithm is similar to the one in [AO94].

We restrict the reset algorithm to be initiated by one and only one initiator, called i. We call this rooted reset algorithm $\mathcal{R}_i^{\mathcal{P}}$. Process i uses Algorithm $\mathcal{PIF}$ as follows: (a) upon a request, i initiates the broadcast of an "abort" message and stops its local execution of $\mathcal{P}$. (b) Upon switching to the broadcast phase, the non-initiators also abort their local execution of $\mathcal{P}$. (c) All the processes (including i) reset their variables related to the problem $\mathcal{P}$ when they feedback. (d) Finally, the processes can roll back their local execution of $\mathcal{P}$ only when they stop participating in the PIF-wave corresponding to $\mathcal{R}_i^{\mathcal{P}}$ (cleaning phase). Since Algorithm

$\mathcal{PIF}$ is snap-stabilizing, all the processes will receive the abort message. After starting $\mathcal{R}_i^{\mathcal{P}}$, the system reaches, in a finite time, a configuration where all processes have their variables reset (precisely right after i has switched to the feedback phase, as explained before). At that point the system is guaranteed to be in a normal initial configuration.

2.4.3 Snap-Stabilizing Snapshot

The goal of a snapshot is to gather at its initiator data which give a *representation* of the configuration of the system. This representation consists of a collection of histories, one per process. Each history consists of a local state and messages received by the process. The computed representation should be consistent to help in verifying the coherence of the system, *e.g.*, to deduce whether or not the configurations of the system are normal. Our solution, noted $\mathcal{S}^{\mathcal{P}}$, is an adaptation in the atomic-state model of the snapshot algorithm provided in [CL85] written in the message-passing model. So, before we present our algorithm, we briefly describe the main idea of the snapshot algorithm in [CL85] applied on a problem $\mathcal{P}$. In this algorithm, a process initiates the snapshot by first recording its state in $\mathcal{P}$ (including its own identifier) and then broadcasting a marker to all its neighbors. Upon receiving a marker, a process becomes marked, records its own state in $\mathcal{P}$ (including its own identifier), and sends the marker to all its neighbors, and so on so. Then, a marked process records every $\mathcal{P}$'s message received from each given neighbor (together with the incoming channel number) until it receives a marker from that neighbor. Once it has received markers from all its neighbors, the process completes its participation in the snapshot algorithm by sending a report (recorded state and messages) to the initiator. In our model (the atomic-state model), the messages are modeled by the ability of a process to read the state of its neighbors. So, the collection of the recorded messages at process p from a neighbor q (in the message-passing model) is replaced by the history of change of state of q after the marking event of p and before that of q.

The aim of Algorithm $\mathcal{S}^{\mathcal{P}}$ is to gather at the initiator the histories of local states of each process from the beginning to the end of its participation to the snapshot (actually a PIF-wave). In our solution, we use a local stack at each process to store its history. Upon the receipt of a broadcast message, a process p resets its stack to its current local configuration. This local configuration is constituted by the state of p and its neighbors (the variables related to the problem $\mathcal{P}$ plus the PIF variables). Then, at each move, the new local configuration is pushed onto the stack until the feedback. During the feedback phase, the process pushes its current configuration on the stack for the last time and merges its own report (actually its stack) and that of its children (in the tree built during the broadcast phase) in a report variable. As Algorithm $\mathcal{PIF}$ ensures that any process executes the broadcast and the feedback phase, the report variable of the initiator includes the report of that process at the end of the feedback phase. At least as much data as the snapshot algorithm of [CL85] are saved during the execution of Algorithm $\mathcal{S}^{\mathcal{P}}$. Hence, we obtain a consistent snapshot when Algorithm $\mathcal{S}^{\mathcal{P}}$ terminates.

2.4.4 Snap-Stabilizing Termination Detection

The *termination detection* can be (maybe not efficiently) performed using any snapshot algorithm. Using $\mathcal{S}^{\mathcal{P}}$, our termination detection algorithm just needs to check the reports at the end of any snapshot. For any snapshot started after the termination of the algorithm $\mathcal{A}$, the report of any process consists of two identical configurations: the current local configurations at the broadcast and the feedback, in that order. In the following, we denote by $\mathcal{TD}^{\mathcal{A}}$ the termination detection algorithm for the algorithm $\mathcal{A}$.

2.5 Overview of the Transformer

Let $\mathcal{A}$ be a distributed algorithm designed for any identified network of arbitrary topology which is neither self- nor snap- stabilizing. Our goal is to transform $\mathcal{A}$ into a snap-stabilizing algorithm using the previous snap-stabilizing solutions (reset, snapshot, ...) as basic building blocks. First, we assume that each process knows an upper bound on the number of processes in the network. Then, following [KP93], we assume that (1) we can define for $\mathcal{A}$ a predicate `OK` which characterizes the normal configurations of the system (this predicate can be computed by a snapshot); and (2) the dynamic specification of $\mathcal{A}$ satisfies the Kleene-closed condition.

Since we consider dynamic specifications, where starting actions are explicit in the safety conditions, their execution, by one or more initiators, is critical in the snap-stabilizing context. As a consequence, the difference between the mono- and multi-initiator cases is important here, while it is simply irrelevant in the Katz and Perry's transformer [KP93]. So, we now present the main principles of our transformer by considering two classes of algorithms:

(a) The algorithms that have a unique initiator. Precisely, in this class, each instance of a given algorithm can be associated to a unique initiator and instances of the same algorithm cannot be merged.

(b) The algorithms that can be initiated by several processes.

Notice that, in the case where $\mathcal{A}$ solves an infinite problem such as the token circulation, the Kleene-closed condition ensures that every execution of $\mathcal{A}$ can be split into an infinite repetition of finite computations, each of them being initiated by a starting action. To obtain the infinite behavior, we just need that the request for the next computation immediately occurs at the completion of the current one. For example, the depth-first token circulation can be seen as successive depth-first sequential traversals of the network.

2.5.1 The Mono-initiator Case

Assume Algorithm $\mathcal{A}$ has a unique initiator i. A simple attempt to transform $\mathcal{A}$ into a snap-stabilizing algorithm is to reset the network (using $\mathcal{R}_i^{\mathcal{A}}$) before starting $\mathcal{A}$. So, we add $\mathcal{R}_i^{\mathcal{A}}$ as an header of $\mathcal{A}$ so that $\mathcal{A}$ starts only after $\mathcal{R}_i^{\mathcal{A}}$ terminates. We denote this new algorithm by $\mathcal{R}_i^{\mathcal{A}} \cdot \mathcal{A}$ and the starting action of $\mathcal{R}_i^{\mathcal{A}} \cdot \mathcal{A}$ is the one of $\mathcal{R}_i^{\mathcal{A}}$. Thus, starting $\mathcal{A}$ now implies first running $\mathcal{R}_i^{\mathcal{A}}$ and then $\mathcal{A}$. As the system has been reset for $\mathcal{A}$ in a snap-stabilizing manner, obviously, $\mathcal{A}$ behaves as in a non-faulty situation, *i.e.*, according to its specification.

Assume now that no starting action of $\mathcal{R}_i^{\mathcal{A}}$ was executed after a fault occurred, but $\mathcal{R}_i^{\mathcal{A}} \cdot \mathcal{A}$ is still running. Then, there is no guarantee that $\mathcal{R}_i^{\mathcal{A}} \cdot \mathcal{A}$ terminates. Hence, after a request, we cannot ensure that the execution will contain at least one starting action of $\mathcal{R}_i^{\mathcal{A}} \cdot \mathcal{A}$. Since $\mathcal{R}_i^{\mathcal{A}}$ is snap-stabilizing, we know that $\mathcal{R}_i^{\mathcal{A}}$ eventually terminates. So, the reason why $\mathcal{R}_i^{\mathcal{A}} \cdot \mathcal{A}$ may not terminate is due to the fact that $\mathcal{A}$ may not terminate (recall that $\mathcal{A}$ is not even self-stabilizing). To prevent the system from this kind of deadlock or livelock, a checking procedure must be added to detect whether the system is in an abnormal configuration and, in this case, abort the current execution before starting a new one. The checking procedure consists in taking snapshot of the system regularly (using the snap-stabilizing snapshot algorithm $\mathcal{S}_i^{\mathcal{A}}$). We use the snapshots to compute two predicates: `OK` which characterizes the normal configurations of the system and `TD` which determines if $\mathcal{A}$ is terminated.[5] Process i now waits until $\mathcal{R}_i^{\mathcal{A}}$ terminates and $\mathcal{S}_i^{\mathcal{A}}$ returns either $\neg$`OK` or `TD`, before starting $\mathcal{R}_i^{\mathcal{A}} \cdot \mathcal{A}$.

[5] As explained in Section 2.4.4, we can use Algorithm $\mathcal{S}_i^{\mathcal{A}}$ for termination detection.

In order to design the checking procedure without fairness assumptions (*i.e.,* under an unfair daemon), we must not execute snapshots concurrently with $\mathcal{A}$. Otherwise, an unbounded number of steps of $\mathcal{A}$ can be executed before the completion of the first snapshot. Hence, we have chosen to schedule at most one step of $\mathcal{A}$ per process, using a $\mathcal{PIF}_i$-wave, before executing a snapshot of the system. After that, if the configuration is a normal one (*i.e.*, if $\mathcal{S}_i^{\mathcal{A}}$ returns OK), we repeat the procedure until $\mathcal{A}$ terminates (*i.e.*, until $\mathcal{S}_i^{\mathcal{A}}$ returns TD). Otherwise, the current execution can be aborted so that i can start $\mathcal{R}_i^{\mathcal{A}} \cdot \mathcal{A}$. Hence, the checking procedure ensures that $\mathcal{R}_i^{\mathcal{A}} \cdot \mathcal{A}$ eventually starts, but without risking to abort any normal execution of $\mathcal{A}$.

The checking procedure is designed as follows: $\mathcal{PIF}_i^{\mathcal{A}}$-waves and $\mathcal{S}_i^{\mathcal{A}}$-waves are performed in sequence until the predicate $\neg\texttt{OK} \vee \texttt{TD}$ is satisfied (at i). During each wave of $\mathcal{PIF}_i^{\mathcal{A}}$, any process can execute at most one action of $\mathcal{A}$: upon receiving a broadcast wave. Since $\neg\texttt{OK} \vee \texttt{TD}$ is satisfied, i can start $\mathcal{R}_i^{\mathcal{A}} \cdot \mathcal{A}$ without risking to abort a normal execution of $\mathcal{A}$. Indeed, either the behavior is fuzzy ($\neg$OK) or the previous execution of $\mathcal{A}$ is terminated (TD).

Hence, we obtain a new algorithm $\texttt{TRANS1}_{\mathcal{A}}$ whose executions just consist of sequences of $\mathcal{PIF}$-waves. The fact that Algorithm $\mathcal{PIF}$ accepts the distributed unfair daemon ensures that $\texttt{TRANS1}_{\mathcal{A}}$ is a snap-stabilizing version of Algorithm $\mathcal{A}$ working under a distributed unfair daemon.

2.5.2 The Multi-initiator Case

Assume now that $\mathcal{A}$ is multi-initiator. We first transform $\mathcal{A}$ into a snap-stabilizing algorithm working under a distributed *weakly fair* daemon. We then explain how to make the transformed algorithm working under a distributed *unfair* daemon. Unlike the mono-initiator case, an initiator cannot unilaterally reset the system before starting $\mathcal{A}$. Indeed, such a reset may interrupt an execution started by some other initiator. The idea here is to let execute $\mathcal{A}$ by all processes to prevent deadlocks. However, we should carefully address the execution of starting actions. So, we precede the initialization of a process i in $\mathcal{A}$ by the execution of the header $\mathcal{H}_i^{\mathcal{A}}$. As in the mono-initiator case, the initialization of the transformed algorithm $\mathcal{H}_i^{\mathcal{A}} \cdot \mathcal{A}$ is the one of $\mathcal{H}_i^{\mathcal{A}}$. More precisely, upon receiving a request, a process i waits until it is able to start $\mathcal{A}$. Then, it initializes $\mathcal{H}_i^{\mathcal{A}}$ and delays the initialization of $\mathcal{A}$ until the completion of $\mathcal{H}_i^{\mathcal{A}}$.

The goal of the header $\mathcal{H}_i^{\mathcal{A}}$ is to check whether the system is in a normal configuration. If not, the system is reset. So, at the termination of $\mathcal{H}_i^{\mathcal{A}}$, the system is in a normal configuration, and i can start $\mathcal{A}$.

Now, to be consistent, the snapshot and the reset of the system must be exclusively executed by a single process. So, the first part of $\mathcal{H}_i^{\mathcal{A}}$ consists of waking up the leader, using a $\mathcal{PIF}_i$-wave, so that it performs these two algorithms. Then, the leader precedes the execution of these two algorithms by a checking snapshot-based procedure in order to be sure that it will execute them alone. Finally, the header $\mathcal{H}_i^{\mathcal{A}}$ terminates when i is informed by the leader that its work is done. Hence, at the termination of $\mathcal{H}_i^{\mathcal{A}}$, the system is in a normal configuration and i can initialize $\mathcal{A}$.

However due to the arbitrary initial configuration, an execution of $\mathcal{A}$ which has not been properly initiated can lead the system to a deadlock or a livelock. So, to prevent such a situation we use another algorithm to control of the correctness of the configurations. As previously, this control has to be executed by the leader alone. But, it has to be implemented at each process since the leader is *a priori* unknown. Thus, the first action is to check whether the process is the actual leader. In that case, the leader executes a snap-stabilizing snapshot to check the correctness of the configuration, and performs a reset if the system is in an abnormal configuration.

Hence, the transformed version, $\texttt{TRANS2}_{\mathcal{A}}$, is snap-stabilizing w.r.t. the specification of the problem solved by $\mathcal{A}$ under the distributed weakly fair daemon only.

Let us now consider the fairness issues. We borrow the same ideas as in [BGJ01, BGJ07]. We first construct an algorithm called $\mathcal{TF}$ which is *total fair* [BGJ07] in any identified connected network, meaning

that each of its executions under the distributed unfair daemon contains an infinitely many actions at each process. Then, we compose $\texttt{TRANS2}_{\mathcal{A}}$ with $\mathcal{TF}$ using the *cross-over composition* [BGJ01] (see below for details). The resulting algorithm, noted $\texttt{TRANS2}_{\mathcal{A}} \diamond \mathcal{TF}$, satisfies the same safety properties as $\texttt{TRANS2}_{\mathcal{A}}$. Moreover, it enforces the liveness properties of $\texttt{TRANS2}_{\mathcal{A}}$ in such a way that they now endure a distributed unfair daemon. Hence, we obtain our final result: $\texttt{TRANS2}_{\mathcal{A}} \diamond \mathcal{TF}$ is snap-stabilizing under the distributed unfair daemon w.r.t. the specification of the problem solved by $\mathcal{A}$.

To construct the total fair algorithm $\mathcal{TF}$ we proceed as follows. We endow each process i with one instance of Algorithm $\mathcal{PIF}$, noted $\mathcal{PIF}_i$. Each process i executes infinitely many $\mathcal{PIF}_i$-waves in sequence. Notice that for every i, j such that $i \neq j$, instances of $\mathcal{PIF}_i$ and $\mathcal{PIF}_j$ run in parallel. Now, since

- the number of processes is finite,
- there are exactly as many instances of $\mathcal{PIF}$ as processes which run at a time (each instance $\mathcal{PIF}_i$ executes its own waves in sequence), and
- each complete wave of each instance $\mathcal{PIF}_i$ is executed within a finite number of steps,

we can deduce that at least one instance $\mathcal{PIF}_i$ performs infinitely many complete $\mathcal{PIF}$-waves in any execution assuming a distributed unfair daemon. By definition, each complete $\mathcal{PIF}$-wave involves all processes. Consequently, $\mathcal{TF}$ is a total fair algorithm.

By composing our transformer $\texttt{TRANS2}_{\mathcal{A}}$ with $\mathcal{TF}$ using the cross-over composition, we obtain a transformer which works assuming a distributed unfair daemon. Indeed, the *cross-over* composition has been designed as a tool for scheduler management. Informally, in the composition $A \diamond B$, the actions of A are synchronized with actions of B: the actions of A are performed only when an action of B is performed too, in the same step. So, the execution of A is fully driven by B, *i.e.*, B acts as a scheduler for A. Hence, if B is a total fair algorithm, then in any execution of $A \diamond B$, every process which is continuously enabled w.r.t. A executes an action of A within finite time, and we are done.

Hence, with $\texttt{TRANS2}_{\mathcal{A}} \diamond \mathcal{TF}$, we obtain that $\mathcal{A}$ can be transformed into a snap-stabilizing algorithm for the same specification working under a distributed unfair daemon.

2.6 Further Implications

Similarly to [KP93], the transformer we propose here is quasi-independent from the computational model. Indeed, our transformer can be achieved using any snap-stabilizing PIF algorithm implemented in the considered model.

We have addressed the snap-stabilization in the more practical message passing system, yet assuming a fully-connected identified network [DDNT10]. In that paper, we have first established that many nontrivial specifications do not admit a snap-stabilizing solution, even assuming infinite memory at each process, in message-passing systems with unbounded yet finite capacity FIFO channels. Now, many such specifications admit self-stabilizing solutions using an infinite memory per process [KP93]. More precisely, we have introduced the notion of *safety-distributed* specifications and shown that no problem having such a specification admits a snap-stabilizing solution in message-passing systems with finite yet unbounded capacity FIFO channels. Intuitively, safety-distributed specification has a safety property that depends on the behavior of more than one process. That is, certain process behaviors may satisfy safety if done sequentially, while violate it if done concurrently. For example, in mutual exclusion, any requesting process must eventually execute the critical section but if after entering the critical section, a process does not execute it alone, then the safety is violated. Roughly speaking, the class of *safety-distributed* specifications includes all distributed problems

where processes need to exchange messages in order to preclude any safety violation. On the other hand, we have proposed a snap-stabilizing PIF and a mutual exclusion algorithm for a fully-connected identified network [DDNT10] where links are FIFO and of known bounded capacity.

After this preliminary work, Levé *et al.* [LMV16] have proposed a snap-stabilizing PIF working in arbitrary connected rooted message passing systems, still assuming FIFO links with known bounded capacity. Hence, our transformer together with the work of Levé *et al.* [LMV16] imply that snap-stabilization is power equivalent to self-stabilization in message passing systems when the links are both FIFO and of known bounded capacity.

Besides, even if the question of expressiveness is closed, we still need to investigate snap-stabilization in various models, in particular the message passing one, to find efficient solutions dedicated to important distributed computing problems such as the token circulation, for example.

Chapter 3

Concurrency in (Self-Stabilizing) Resource Allocation Algorithms

This chapter is a digest of several works [DDHL11, CDDL15, ADD17b] we have done on resource allocation problems.

3.1 Resource Allocation Problems

One of the main goals in distributed systems is to share resources (*e.g.*, printers, computing servers, tape driver) among a large number of processes. It is commonly assume that the number of available resources is drastically smaller than the number of processes.

Recall that access to the resources is controlled using a special portion of code at each process, called the critical section. Each process may need to execute its critical section, which is assumed to last a finite, yet unbounded, time. For example, the well-known *mutual exclusion* problem [Lan77] consists in organizing fair access of all requesting processes to a single non-shareable reusable resource, *e.g.*, a printer or a tape driver.

Along the years, many other resource allocation problems have been studied, *i.e.*, ℓ-exclusion [FLBB79], k-out-of-ℓ-exclusion [Ray91], dining philosophers [Dij78], drinking philosophers [CM84]. Resource allocation problems can be discriminated according to several parameters, *e.g.*, the number of resources that can be requested by a process, the global number of resources, the safety property to ensure (either global, or local, *i.e.*, related to the closed neighborhood), the different types of resources; see examples in the table below.

	size of requests	**number of resources**	**safety**	**number of resource types**
ℓ-exclusion	1	ℓ	Global	1
k-out-of-ℓ-exclusion	$[1..k]$ $(k \leq \ell)$	ℓ	Global	1
dining philosophers	1	1 for each pair of neighbors	Local	1
drinking philosophers	$[1..\delta(p)]$ ($\delta(p)$ is the degree of p)	1 for each pair of neighbors	Local	the number of edges

3.2 Efficiency in Resource Allocation Problems

3.2.1 Availability

Classically, efficiency of resource allocation algorithms is evaluated by the *waiting time*, *i.e.*, the maximum time for a requesting process to enter the critical section. Actually, the waiting time captures the resource *availability* offered by the algorithm. In other words, the smaller is the waiting time, the faster the resource is available for the requesting process.

Notice that, there exist alternative definitions of the waiting time. For example, the waiting time is sometimes defined as the maximum number of times processes enter the critical section between the request and the entrance of a process p in the critical section. We do not consider such a definition here.

3.2.2 Concurrency

The mutual exclusion problem is inherently sequential since no two processes should access this resource concurrently. However, there are many other resource allocation problems which, in contrast, allow several resources to be accessed simultaneously. In those problems, parallelism on access to resources may be restricted by some of the following conditions:

1. The maximum number of resources that can be used concurrently, *e.g.*, the *ℓ-exclusion* problem [FLBB79] is a generalization of the mutual exclusion problem which allows use of ℓ identical copies of a non-shareable reusable resource among all processes, instead of only one in standard mutual exclusion.

2. The maximum number of resources a process can use simultaneously, *e.g.*, the *k-out-of-ℓ exclusion* problem [Ray91] is a generalization of ℓ-exclusion where a process can request for up to $k \leq \ell$ resources simultaneously.

3. Some topological constraints, *e.g.*, in the *dining philosophers* problem [Dij78], two neighbors cannot use their common resource simultaneously.

For efficiency purposes, algorithms solving such problems must be as parallel as possible. As a consequence, these algorithms should be, in particular, evaluated at the light of the level of *concurrency*[1] they permit (*i.e.*, their ability to maximize the utilization rate of resources), and this level of concurrency should be captured by a dedicated property. However, most of the resource allocation problems are specified in terms of safety and fairness properties only, *i.e.*, most of them include no property addressing concurrency performances; see *e.g.* [BPV04, CDP03, GH07, Hua00, NA02b]. Now, as quoted by Fischer *et al.* [FLBB79], specifying resource allocation problems without including a property of concurrency may lead to degenerated solutions, *e.g.*, any mutual exclusion algorithm realizes safety and fairness of ℓ-exclusion. Indeed, fairness is the same in both problems, while the safety of ℓ-exclusion is weaker than that of mutual exclusion: if a process p enters the critical section, then the overall number of processes executing the critical section concurrently (including p) is at most ℓ.

To address this issue, Fischer *et al.* [FLBB79] proposed an *ad hoc* property to capture concurrency in ℓ-exclusion. This property is called *avoiding ℓ-deadlock* and is informally defined as follows: "if fewer than ℓ processes are executing their critical section, then it is possible for another process to enter its critical section, even though no process leaves its critical section in the meantime."

To clearly understand this notion of avoiding ℓ-deadlock, we now describe an example of solution that does not satisfy this property. In airports, there are usually many aligned registration desks (say ℓ) and access

[1] Concurrency properties are liveness properties.

of clients to those desks is organized as a single queue along a serpentine belt. A client enters its "critical section" when he/she is at the head of the queue and at least one desk is free: he/she moves to a free desk. Assume now that the client at the head of the queue is quite sleepy or just focused on reading a new paper, then all other clients are blocked until he/she wakes up and leaves the queue, even if several (more than one) desks are free. Clearly, this way of managing registration desks (actually, the usual one) does not satisfy the avoiding ℓ-deadlock property.

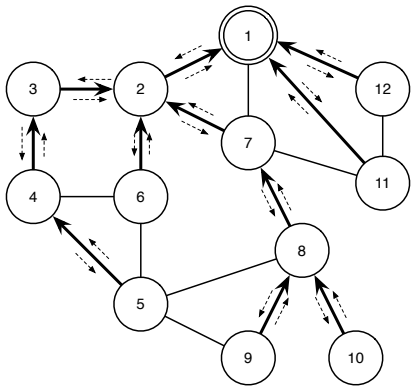


Figure 3.1: Euler tour of a spanning tree given by the dashed arrows. The number inside each node gives its preorder ranking in the tour. An Euler tour actually corresponds to a DFS traversal of the spanning tree, starting from the root.

We should underline that it is easy to design a ℓ-exclusion token-based algorithm that satisfies the ℓ-deadlock property. Consider for example an oriented ring topology (this example can be easily generalized to arbitrary connected networks by computing a spanning tree and then use the virtual embedded ring defined by its Euler tour;[2] see Figures 3.1 and 3.2). Then, the ℓ tokens circulate in the ring clockwise. Upon receiving a token, if the process is requesting, then it keeps the token and enters the critical section; the token will be released at the end of the section. In all others cases, including the case where the process already holds another token and is executing the critical section, the received token is immediately released. Hence, if less than ℓ tokens are used by some processes for an arbitrary long time, then it is still possible for another process to enter its critical section, since unused tokens circulate until being caught by some requesting process.

3.3 A General Property for Concurrency

After the work of Fischer *et al.* [FLBB79], some other properties, inspired from the avoiding ℓ-deadlock property, have been proposed to capture the level of concurrency in some other resource allocation problems, *e.g.*, k-out-of-ℓ-exclusion [DHV03b] and committee coordination [BDP16]. However, all these properties are specific to a particular problem, *e.g.*, the avoiding ℓ-deadlock property cannot be applied to committee coordination. In [ADD17b], we have proposed to generalize the definition of avoiding ℓ-deadlock to any resource allocation problems. We call this new property the *maximal concurrency*.

[2] A cycle that traverses each edge of the networks exactly once in each direction.

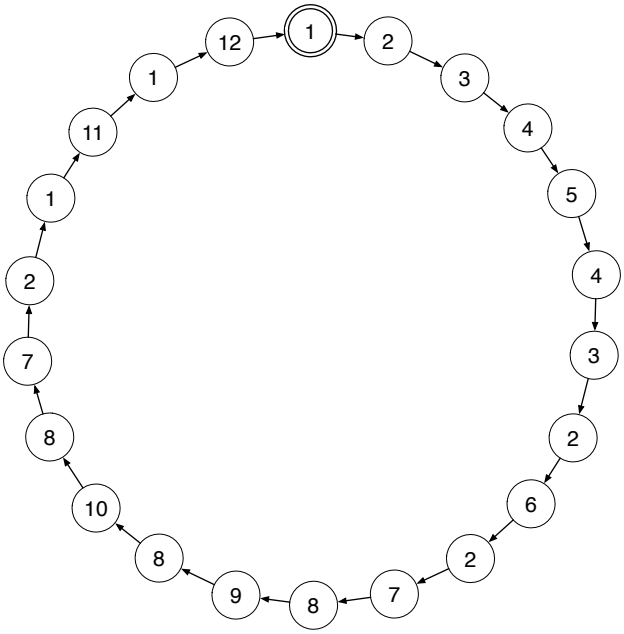


Figure 3.2: The virtual embedded ring defined by the Euler tour proposed in Figure 3.1.

3.3.1 Definition of Maximal Concurrency

Informally (refer to [ADD17b] for the formal definition), maximal concurrency can be defined as follows: if there are processes that can access the resources they are requesting without violating the safety of the considered resource allocation problem, then at least one of them should eventually satisfies its request, even if no process releases the resource(s) it holds meanwhile. In other words, as long as it is possible, requests should be satisfied regardless of the duration of critical sections.

More precisely, let P_{free} be the set of requesting processes that can enter their critical section without violating the safety of the considered problem. If P_{free} is not empty, then a requesting process should eventually enters its critical section, even if no process leaves its critical section meanwhile. Equivalently, if we block processes forever in critical section,[3] then P_{free} is eventually empty. It is important to note that a process p may leave P_{free} without accessing its critical section, *e.g.*, another process q enters the critical section and while q is in its critical section, p cannot enter its critical section without violating the safety.

For example, in the ℓ-exclusion problem, P_{free} is the set of all requesting processes if there are less than ℓ processes executing their critical section, $P_{free} = \emptyset$ otherwise. This instantiation of maximal concurrency is equivalent to the avoiding ℓ-deadlock property.

3.3.2 Maximal Concurrency in k-out-of-ℓ-exclusion

Recall that the k-out-of-ℓ exclusion problem is a generalization of the ℓ-exclusion problem where each process can request up to $k \leq \ell$ identical units of a non-shareable reusable resource.

[3]This assumption is only used to express the condition "even if no process releases the resource(s) it holds meanwhile" in the definition of maximal concurrency, all other properties (safety and fairness) are established assuming finite critical sections. An alternative more formal definition that does not use this artifact is proposed in [ADD17b].

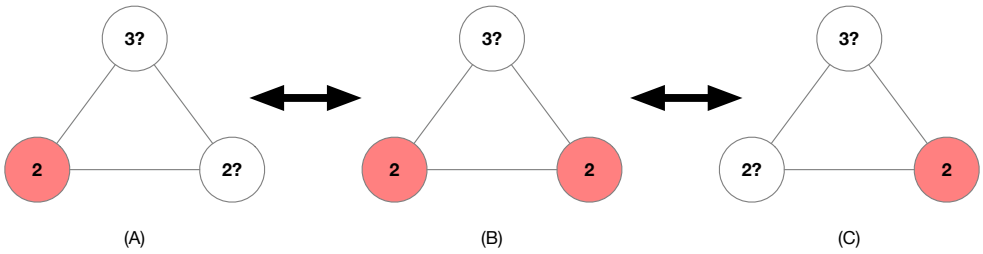


Figure 3.3: Impossibility of the strict $(3,4)$-liveness.

Let X be the number of available resources, *i.e.*, ℓ minus the sum of currently used resources. Then, P_{free} is the set of processes that requests at least one but at most X resources. Using this instantiation, we obtain a definition of maximal concurrency which is equivalent to the *strict* (k,ℓ)*-liveness* property defined by Datta *et al.* [DHV03a],[4] which basically means that if *at least one* request can be satisfied using the available resources, then eventually one of them is satisfied, even if no process releases resources in the meantime.

In [DHV03a], the authors show the impossibility of designing a k-out-of-ℓ exclusion algorithm satisfying the strict (k,ℓ)-liveness. To see this, consider Figure 3.3 and assume $k=3$ and $\ell=4$. The system may eventually reaches Configuration (A). Now, from that configuration, the process requesting two resources (the rightmost one) should be able to enter the critical section, even if the leftmost process does not leave its critical section meanwhile. Indeed, this is the only process that can satisfy its request with the available resources (two) without violating safety. This leads to Configuration (B). Assume now that the first process to leave its critical section from (B) is the leftmost one. Then, assume that this latter process immediately requests again two resources. This leads to configuration (C) which is symmetric to Configuration (A). Hence, to ensure the strict $(3,4)$-liveness, we obtain a possible execution where a process (the one requesting three resources) never enters in its critical section, violating the fairness property, a contradiction.

To circumvent this impossibility, Datta *et al.* have proposed a weaker property simply called (k,ℓ)-*liveness*, which means that if *any* request can be satisfied using the available resources, then eventually one of them is satisfied, even if no process releases resources in the meantime. Despite this property is weaker than maximal concurrency, it can be expressed using our formalism as follows. Let X be the number of available resources. Then, $P_{free}=\emptyset$, if there is a process that requests more than X resources, otherwise P_{free} is the set of all requesting processes. This might seem surprising, but observe that in this latter case, the set P_{free} is distorted from its original meaning.

Actually, we have proposed in [DDHL11] a self-stabilizing k-out-of-ℓ algorithm in message passing for rooted tree networks that satisfies this latter property. By composing our algorithm with a spanning tree construction, we obtain a solution working in arbitrary connected rooted networks. This solution is token-based: the tokens circulate forever along the virtual embedded ring defined by the Euler tour of the tree. Moreover, our algorithm is based on two modules: a non self-stabilizing token-based k-out-of-ℓ algorithm and a self-stabilizing controller that regulate the number of tokens in the system (inspired from [HV01]).

Let's start with the non self-stabilizing solution. The naive idea is to make ℓ resource tokens circulate along the virtual embedded ring defined by the Euler tour of the tree (following the same sense of direction). A process catches a token when it is requesting and enters the critical section when it holds enough tokens. Now, this greedy approach does not work: it can lead to a deadlock where each token is held by a process that waits for more tokens to satisfy its request. To remove such a deadlock, we introduce an additional special token called the *pusher*. This token is not a resource. It circulates like the other tokens, but upon receiving the

[4]The proof of equivalence is available in [ADD17b].

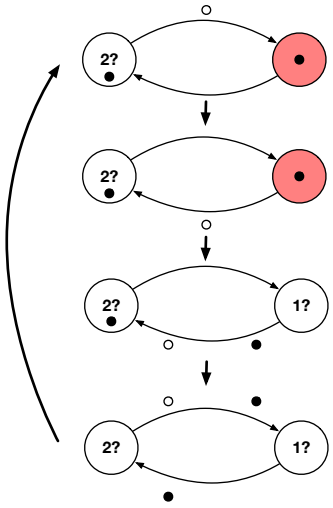


Figure 3.4: Example of possible livelock with $k = 2$ and $\ell = 2$. The pusher is the white token. Resource tokens are colored in black. Processes executing the critical section are colored in red.

pusher, a process releases all the resource tokens it holds, except if its request is satisfied. However, solving the deadlock may create livelock; see Figure 3.4.

To prevent the system from livelock, we introduce a last special (non-resource) token called the *priority* token. The priority token cancels the effect of the pusher. This token circulates as the previous ones. However, a requesting process keeps the priority and resource tokens even after receiving the pusher until its request is satisfied.

To make this solution self-stabilizing, we compose it with a controller module which regulates the number of tokens in the systems. Again, the controller is implemented using another perpetual token circulation. However, this time, the circulation is a self-stabilizing one. For example, one can use the counter flushing technique of Varghese [Var00] (to that goal, it should be additionally assumed that the link capacity is bounded and such a bound should be known by all processes). We use the new token as a flag. At the beginning of a Euler tour traversal, the root resets a counter to 0 for each token type, except (of course) the controller token. Each time the root receives a token which is not the controller, it checks whether the corresponding counter has already reached its maximum expected value, and if so, the token is discarded. Otherwise, the root simply increments the counter and uses the received token following the non-self-stabilizing algorithm. Moreover, at the end of the traversal, the root creates tokens if some of them are missing.

3.3.3 Concurrency in LRA

In [ADD17b], we have considered the maximal concurrency in the context of the *Local Resource Allocation (LRA)* problem, defined by Cantarell *et al.* [CDP03]. LRA is a generalization of resource allocation problems in which resources are shared among neighboring processes. Dining philosophers, local readers-writers, local mutual exclusion, and local group mutual exclusion are particular instances of LRA. In contrast, local ℓ-exclusion and local k-out-of-ℓ-exclusion cannot be expressed with LRA although they also deal with neighboring resource sharing.

3.3.3.1 Local Resource Allocation

In the *local resource allocation* (LRA) problem, each process requests at most one resource at a time. We denote by $\mathcal{R}_p$ the set of resources a process p may access. The problem is based on the notion of compatibility: two resources X and Y are said to be *compatible* if two neighbors are allowed to access them concurrently. Otherwise, X and Y are said to be *conflicting*. In the following, we denote by $X \rightleftharpoons Y$ (resp. $X \not\rightleftharpoons Y$) the fact that X and Y are compatible (resp. conflicting). Notice that $\rightleftharpoons$ is a symmetric relation.

Using the compatibility relation, the LRA problem consists in ensuring that every process which requires a resource r eventually accesses r while no other conflicting resource is currently used by a neighbor. In contrast, there is no restriction for concurrently allocating the same resource to any number of processes that are not neighbors.

Notice that the case where there are no conflicting resources is trivial: a process can always use a resource whatever the state of its neighbors. So, from now on, we will always assume that there exist at least one conflict, *i.e.*, there are (at least) two neighbors p, q and two resources X, Y such that $X \in \mathcal{R}_p, Y \in \mathcal{R}_q$ and $X \not\rightleftharpoons Y$. This also means that any network considered from now on contains at least two processes.

Note also that the term "resource" is used here in a broad sense. It actually encompasses various realities, depending on the context. In some applications, like in [GT07], the resource can be “virtual”, while in some other the resource is a physical device. Furthermore, in some cases, a resource actually represents a pool of resources of the same type. In this latter case, the pool of resources consists of multiple anonymous resources.

Specifying the relation $\rightleftharpoons$, it is possible to define some classic resource allocation problems in which the resources are shared among neighboring processes, as shown below.

Example 1: Local Mutual Exclusion. In the *local mutual exclusion* problem, no two neighbors can concurrently access the unique resource. So, there is only one resource X common to all processes and $X \not\rightleftharpoons X$.

Example 2: Local Readers-Writers. In the *local readers-writers* problem, the processes can access a file in two different modes: a read access (the process is said to be a *reader*) or a write access (the process is said to be a *writer*). A writer must access the file in local mutual exclusion, while several reading neighbors can concurrently access the file. We represent these two access modes by two resources at every process: R for a “read access” and W for a “write access.” Then, $R \rightleftharpoons R$, but $W \not\rightleftharpoons R$ and $W \not\rightleftharpoons W$.

Example 3: Local Group Mutual Exclusion. In the *local group mutual exclusion* problem, there are several resources $r_0, r_1, \ldots, r_k$ shared between the processes. Two neighbors can access concurrently the same resource but cannot access different resources at the same time. Then:

$$\forall i \in \{0,\ldots,k\}, \forall j \in \{0,\ldots,k\}, \begin{cases} r_i \rightleftharpoons r_j & \text{if } i = j \\ r_i \not\rightleftharpoons r_j & \text{otherwise} \end{cases}$$

3.3.3.2 Maximal Concurrency in LRA

We have shown that maximal concurrency cannot be achieved in the wide class of instances of the LRA problem, where every process can request the same set of resources $\mathcal{R}$ (*i.e.*, $\forall p \in V, \mathcal{R}_p = \mathcal{R}$) and $\exists x \in \mathcal{R}$ such that $x \not\rightleftharpoons x$. Notice that the local mutual exclusion and the local readers-writers problem belong to this class of LRA problems.

Basically, the impossibility proof follows the same scheme as for the impossibility of strict (k,ℓ)-liveness in k-out-of-ℓ-exclusion. If the maximal concurrency is achieved, then it is possible to construct an execution where the neighbors of some process p alternatively execute their critical section using a resource that is in conflict with the one requested by p. In such an execution, p never enters the critical section and so fairness is violated.

3.3.3.3 Partial and Strong Concurrency in LRA

As we have just seen, in LRA, maximal concurrency and fairness are (most of the time) incompatible properties: it is impossible to implement an algorithm achieving both properties together. As unfair resource allocation algorithms are clearly unpractical, we have proposed to weaken the property of maximal concurrency. We have introduced the weak parametric version of maximal concurrency called *partial concurrency*. The aim of this property is to capture the maximal level of concurrency that can be obtained in any instance of the LRA problem without compromising fairness.

Partial concurrency. Recall that maximal concurrency requires that if we block processes forever in their critical section, then P_{free} is eventually empty. Partial concurrency relaxes maximal concurrency in the following sense: if we block processes forever in critical section, then P_{free} is eventually included in a set of processes X such that $\mathcal{P}(X)$ holds ($\mathcal{P}$ being a predefined parameter). Partial concurrency is a generalization of maximal concurrency since we retrieve maximal concurrency by letting $\mathcal{P}(X) \equiv X = \emptyset$. The problem is now to define a predicate as restrictive as possible, yet allowing to solve the LRA.

Strong concurrency. The impossibility proof exhibits a possible scenario for some instances of LRA which shows the incompatibility of fairness and maximal concurrency: enforcing maximal concurrency can lead to unfair behaviors where some neighbors of a process alternatively use resources which are conflicting with its own request. To achieve fairness, we must then relax the expected level of concurrency in such a way that this situation cannot occur indefinitely.

We denote by $\mathcal{CN}_p$ the set of conflicting neighbors of process p in a given configuration, *i.e.*, the neighbors of p that request or use a resource which is incompatible with the one requested by p.

The key idea is that sometimes the algorithm should prioritize one process p against its neighbors, although it cannot immediately enter the critical section because some of its conflicting neighbors are in critical section. In this case, the algorithm should momentarily block all conflicting requesting neighbors of p that can enter critical section without violating safety, so that p enters critical section first. In the worst case, p has only one conflicting neighbor q in critical section and so the set of processes that p has to block contains up to all conflicting (requesting) neighbors of p that are neither q, nor conflicting neighbors of q (by definition, any conflicting neighbor common to p and q cannot access critical section without violating safety because of q). Hence, we can let $\mathcal{P}(X) = \exists p \in V, \exists q \in \mathcal{CN}_p : X = \mathcal{CN}_p \setminus (\{q\} \cup \mathcal{CN}_q)$. We then call *strong concurrency* the corresponding refinement of partial concurrency. Strong concurrency seems to be very close to the maximum degree of concurrency which can be ensured by an algorithm solving all instances of LRA.

A snap-stabilizing LRA algorithm satisfying strong concurrency. The algorithm we have proposed is written in the atomic-state model assuming a distributed weakly fair daemon. Moreover, we assume that the network is connected and the processes have unique identifiers.

We first implement a greedy approach: when several neighboring processes are requesting some resources, the one with the highest identifier has the priority. Moreover, requesting processes that are in conflict with

either a requesting neighbor with an higher identifier, or a neighbor already executing the critical section should not be considered in this greedy approach. Such processes are said to be *blocked*. Each process maintains a Boolean flag to inform its neighbors about whether or not it is blocked. While blocked, the process should be ignored by its neighbors.

Of course, this greedy approach is not fair. Hence, we use a token circulation to enforce fairness: the token holder has priority whatever the values of the identifiers are. Moreover, conflicting neighbors of a requesting token holder should be blocked. If requesting, a process keeps the token until it enters the critical section.

The token circulation we use is a self-stabilizing one. As it is only used to ensure fairness, it is not an issue to obtain a snap-stabilizing solution. By the way, safety is trivial since local. However, during the self-stabilization phase, we may sometimes have two neighbors that hold a token. In this case, we use the identifiers to break ties.

Notice that this algorithm ensures strong concurrency because, if we block forever processes in critical section, then eventually the set of requesting processes X that are blocked, despite being not in conflict with some neighbor executing its critical section, are (if any) conflicting neighbors of the unique token holder. Moreover, if such processes exist, this means that the token holder is requesting, yet in conflict with a neighbor q already executing the critical section. In this case, q and its conflicting neighbors are not part of X.

To evaluate the waiting time of our solution, we assume that each critical section is executed within one round. In that case, the waiting time of our solution is $O(n)$ rounds, where n is the number of processes.

3.4 Concurrency versus Availability

A drawback of the previous solution is its waiting time, which depends on a global parameter (n) while the problem (LRA) is local. This negative result seems to be surprising since one can expect that maximizing the utilization rate of resources (the goal of the concurrency properties) would imply reducing the waiting time (which evaluates the availability of resources). Now, we have shown in [CDDL15] that availability and concurrency are sometimes incompatible goals. More precisely, we have shown that availability and concurrency are orthogonal issues in the ℓ-exclusion problem in a ring network: no algorithm can optimize both properties in that context.

Actually we have shown that any ℓ-exclusion algorithm achieving maximal concurrency in a ring has a waiting time in $\Omega(n-\ell)$ rounds. The idea of the proof is that, due to the unpredictability of the requests, it is possible that at some point, all resources are used by ℓ processes gathered in a limited area of the ring and at that time a process far away becomes requesting. In that case, $\Omega(n-\ell)$ rounds are necessary before the process can enter the critical section since it needs at least that time to be informed that some process has released a resource when it happens.

Actually, this lower bound is counterintuitive with the straightforward idea that when using ℓ copies of the resource, one could expect a waiting time in $O(\frac{n}{\ell})$ rounds, the trivial asymptotic optimal bound. Now, the lower bound clearly states that when ℓ is very small compared to n, a $O(\frac{n}{\ell})$ rounds waiting time is impossible in a ring if the algorithm also ensures maximal concurrency.

Since we already know that maximal-concurrent (self-stabilizing) ℓ-exclusion solutions exist, *e.g.*, [HV01], we have decided to explore whether it is possible to design a non maximal-concurrent ℓ-exclusion that ensures the fastest possible waiting time, *i.e.*, $O(\frac{n}{\ell})$ rounds. Notice that a fast waiting time also prevents degenerated solution, *e.g.*, when ℓ is large, no mutual exclusion algorithm can provide a $O(\frac{n}{\ell})$-round waiting time.

We have proposed in [CDDL15] a ℓ-exclusion algorithm offering a $O(\frac{n}{\ell})$-round waiting time in any arbitrary connected identified network. This algorithm is written in the atomic-state model and assume

a distributed weakly fair daemon. This algorithm is actually a composition of three modules (using the hierarchical collateral composition we have introduced in [DLD+13]). First, it is a composition of the ℓ-exclusion algorithm for rings with a spanning tree algorithm: as previously explained, we emulate the ring algorithm on the virtual embedded ring defined by the Euler tour of the constructed spanning tree. Specifically, the spanning tree construction can be any self-stabilizing one working in arbitrary connected and identified network under the distributed weakly fair daemon. Then, the ring algorithm assumes a rooted and oriented ring. The overall idea of the algorithm is rather simple: we partition the network into ℓ regions, called *segments*, containing $O(\frac{n}{\ell})$ processes each. We make a single token circulating into each region. Thanks to that, every process will receive a token every $O(\frac{n}{\ell})$ rounds. So, each process can access the critical section every $O(\frac{n}{\ell})$ rounds, and the fast waiting time property is achieved. Actually, the ring algorithm is a composition of two layers: a layer which aims at splitting the ring into segments and a layer that makes one token circulating into each segment.

More precisely, to obtain a load-balanced partition into ℓ segments, we proceed as follows:

- If ℓ divides n, then each segment has length $\lfloor \frac{n}{\ell} \rfloor$.
- Otherwise, let $x = n \bmod \ell$. The first x segments will have length $\lceil \frac{n}{\ell} \rceil$, and the remaining $\ell - x$ segments will have length $\lfloor \frac{n}{\ell} \rfloor$.

Then, the self-stabilizing token circulation for each segment is inspired by the snap-stabilizing *Propagation of Information with Feedback* algorithm for rooted tree networks given in [BDPV07]. It can be seen as a token circulation for oriented line networks that uses only three states per process. Notice that the third token algorithm of Dijkstra [Dij74] is also a self-stabilizing token circulation for oriented line networks which uses only three states per process, but is designed for the central daemon, while we need here a solution for a distributed daemon.

With the solution for ring networks, we obtain a waiting time in at most $2\lceil \frac{n}{\ell} \rceil - 2$ rounds: the worst-case being when an extremity, w.r.t. the segment, process becomes requester right after releasing the token. Then, in an arbitrary connected network, we should remark that the virtual embedded ring contains $N = 2n - 2$ nodes (indeed, there are $n - 1$ links in the spanning tree and each of them appears exactly twice in the ring). So, the waiting time is at most $2\lceil \frac{N}{\ell} \rceil - 2$ rounds, *i.e.*, $2\lceil \frac{2n-2}{\ell} \rceil - 2$ rounds.

A shortcoming of our proposal is that nodes whose degree is high in the tree will hold tokens more often than those whose degree is low. Indeed, the number of times a process appear in the virtual embedded ring is its degree in the spanning tree. This problem can be mitigated using the rooted spanning tree construction given in [BF14]. That construction builds a spanning tree whose degree differs at most one from the minimum possible. Consequently, it reduces the gap between low and high degrees. However, an experimental study is still missing to validate this idea. Finally, notice that using the spanning tree algorithm of Blin and Fraignaud [BF14], our solution stabilizes in $O(n)$ rounds using $\Theta(\log n)$ bits per process (under the assumption that a process identifier requires $\Theta(\log n)$ bits).

Other shortcoming of our solution is that it requires the weakly fair assumption. However, it seems to difficult to envision a solution with an asymptotically optimal waiting time without this assumption, since it allows to make the ℓ single token circulation working independently from each other. Hence, we conjecture that we cannot remove this assumption, *i.e.*, there does not exist a solution ensuring an asymptotically optimal waiting time under the unfair daemon.

To conclude, we saw that, in the context of ℓ-exclusion, concurrency (formalized by the avoiding ℓ-deadlock property) and availability (captured by the fast waiting time) are somehow incompatible issues: they can be implemented separately but not together. So, a natural question arises: which one should be implemented? Well, it depends ... When the time spent into critical section is very large compared to

communication time, then resources are rare, so we should try to maximize their utilization rate. Hence in that case, maximizing the concurrency seems to be more appropriate. In contrast, when the time spent into critical section is small as compared to communication time, resources are quickly available and so it seems better to optimize the waiting time.

Chapter 4

New Derived Forms of Self-Stabilization

In this chapter, we present three derived forms of self-stabilization we have introduced; see [DDF10, AD17, ADDP19b]. Recall that the overall goal of such derived forms is to sharply qualify the fault tolerance properties of an algorithm. We also discuss the advantages and drawbacks of these properties. Finally, we replace them into the taxonomy of existing derived forms of self-stabilization.

4.1 Fault-Tolerant Pseudo-Stabilization

This section deals with results given in [DDF10]. In that paper, we introduce *fault-tolerant pseudo-stabilization* as an alternative to *fault-tolerant self-stabilization* proposed in [BK97].

4.1.1 Definition and Comparison with Related Properties

Let first recall the difference between "simple" self- and "simple" pseudo- stabilization; for more details, see page 18. Roughly speaking, regardless the initial configuration of the system, any execution of a self-stabilizing algorithm eventually reaches a legitimate configuration from which it *cannot deviate* from its specification. More precisely, this means that the specification of the algorithm is satisfied in all possible suffix starting from a legitimate configuration. Pseudo-stabilization is weaker in the following sense. Regardless of the initial configuration of the system, any execution of a pseudo-stabilizing algorithm eventually reaches a configuration from which it *does not deviate* from its specification. More precisely, this means that any execution of the algorithm has a correct suffix (*i.e.*, a suffix that satisfies the intended specification).

As explained before, self-stabilization and pseudo-stabilization have been initially introduced to cope with transient failures only. Hence, fault-tolerant self-stabilization (*ftss* for short) simply consists in ensuring self-stabilization in systems prone to both transient failures and process crashes [BK97]. In the same line, we have called fault-tolerant pseudo-stabilization (*ftps* for short) the ability of an algorithm to pseudo-stabilize in presence of both transient failures and process crash failures.

Like for pseudo- and self-stabilization, ftps is weaker than ftss. However, ftps is not comparable with simple self-stabilization. Indeed, on the one hand, ftps allows to handle more complex failure patterns. But, on the other hand, the correctness property guaranteed by a fault-tolerant pseudo-stabilizing algorithm is weaker than that of a self-stabilizing algorithm.

4.1.2 Fault-tolerant Pseudo- and Self- Stabilizing Leader Election in Partially Synchronous Systems

We have investigated ftss and ftps in the context of the leader election problem. Notice that, due to the intrinsic absence of termination detection, the leader election specification in fault-tolerant stabilizing systems is similar to the *eventual leader election* problem in crash-prone non-stabilizing systems, *i.e.*, all correct processes should eventually always designate the same correct process as leader. Eventual leader election is also called *Omega* and is often denoted by Ω. Omega is of prime interest, since it is actually an important failure detector which is necessary to solve the consensus in asynchronous systems. More precisely, Ω is the weakest failure detector to solve consensus in an asynchronous system where a majority of processes is correct [CHT96].

It is important to note that without very strong assumptions, fault-tolerant self-stabilization is usually impossible; see *e.g.*, [BK97]. This general claim remains true for the leader election problem, as we shall see latter. Hence, we had to weaken the studied failure patterns by considering what we called *static failures*. Recall first that a crashed process definitely stops executing its local algorithm. This way a process crash is not a transient fault. Indeed, such a fault is permanent, while a component (link or process) hit by a transient fault eventually recover. Another particular aspect of process crashes is that they erratically occur in the system, *e.g.*, we have no clue about their frequency. More generally, we say that failures are *static* if their frequency is low. Conversely the failures are said to be *dynamic*. An immediate consequence of the static assumption is that all static failures (including crashes) can be treated as transient faults: we can consider as initial an arbitrary configuration where process may be already crashed but from which no further failures may occur. As we will see, the static assumption has an important impact on the solvability of the fault-tolerant self-stabilizing leader election. In contrast, we have shown that an algorithm is pseudo-stabilizing for the leader election in a system with dynamic crashes if and only if it is pseudo-stabilizing for the leader election in the same system, except that crashes are static.

To sum up, we have considered ftss and ftps in fully connected identified systems with arbitrary initialization and static process crashes. Moreover, we have made (when possible) weak assumptions on link reliability (*i.e.*, many links may be lossy). Finally, impossibility results in [AH93, ADGFT08] constraint us to consider partially synchronous systems, *i.e.*, systems where we have guarantees on the message transmission time in all or a part of the links. Indeed, the impossibility results in [AH93, ADGFT08] imply that pseudo-stabilizing leader election cannot be solved in asynchronous systems with fair lossy links and static process crashes. In that context, a link is said to *timely* (resp. *eventually timely*) if there exists (resp. eventually exists) a bound (here known by the processes) on the message transmission time in the link; in particular such a link is reliable (resp. eventually reliable). Notice that the fact that the bound holds immediately (*i.e.*, the link is timely) or eventually (*i.e.*, the link is eventually timely) as no impact on stabilizing systems. Hence, we almost only consider partial synchronous systems with some timely links.

Based on all these assumptions, our goal has been to find the borderline assumptions where we go from the possibility to have fault-tolerant self-stabilizing solutions to the possibility to only have fault-tolerant pseudo-stabilizing ones. We have provided several impossibility results and algorithms, notably by considering the communication-efficient aspect. However, our main result have been to show that fault-tolerant self-stabilizing leader election is impossible to achieve whenever the system does not contain any *eventual timely routing overlay*, *i.e.*, systems where there exists a strongly connected subgraph among the correct nodes made of timely links only. In contrast, there exist many weaker systems where fault-tolerant pseudo-stabilizing leader election can be achieved. Maybe the weakest is the one where there exists at least one timely source (or, equivalently, an eventual timely source), *i.e.*, a correct node whose all outgoing links to other correct nodes are timely (all other links may be asynchronous and lossy). This latter result clearly establish that ftps

is strictly stronger than ftss in terms of expressive power.

To obtain our impossibility results we have extensively used the notion of (locally) *indistinguishable executions*. The overall idea is as follows. Consider an execution where all links behave as timely links for an arbitrary long time. If the algorithm is self-stabilizing, in such an execution a legitimate configuration is eventually reached. Consider now an execution with the same prefix until the legitimate configuration but from which all messages issued by the leader are lost. Since, only outgoing links from the leader are not timely, the system still satisfies the assumption about the existence of a source, for example. Now, if the processes eventually change their leader, this contradicts the fact that the execution has reached a legitimate configuration. On the other hand, if all processes keep the same leader, we can construct an indistinguishable execution starting from the legitimate configuration where the leader is crashed. The execution is locally indistinguishable since all non-leader processes start from the same local state and receive the same messages at the same times. Hence, we also obtain a contradiction in this case.

Our algorithmic solutions extensively used *accusation counters*. The idea is that a counter is attached to each process. The counter of process p should be incremented each time another process q is waiting a message from p for too long. In such a case, q sends an *accusation message* to p, and p increments its counter upon the receipt of such an accusation message. Then, the values of the counters are used to eventually elect a reliable process (we also used identifiers to break ties). Notice that we successfully adapt this technique to implement efficient routing algorithms in WSNs; see [ADLP11, ADJL17].

4.1.3 Drawbacks and Possible Extensions

There are two important drawbacks for the fault-tolerant pseudo-stabilizing solutions we proposed in [DDF10]. First, almost all of them assume unbounded process memories. The possibility of designing bounded process memory solutions in weak partially synchronous systems is still today an open question. Second, by essence, the time before reaching a correct suffix in any execution of a pseudo- yet non self- stabilizing algorithm cannot be bounded; see Section 1.5.1.1. This makes the study of *speculation* [KAD$^+$09] attractive in such systems. Indeed speculation consists in guaranteeing that the system satisfies its requirements for all executions (here, the fact of being pseudo-stabilizing), but also exhibits significantly better performances (here, the fact that the correct suffix can be reached in bounded time) in a subset of more probable executions (*e.g.*, in stronger partially synchronous systems). The main idea behind speculation is that the worst possible scenarios (*e.g.*, full asynchrony) are often rare (even unlikely) in practice. So, a speculative algorithm is assumed to self-adapt its performances w.r.t. the "quality" of the environment, *i.e.*, the more favorable the environment is, the better the complexity of the algorithm should be.

4.2 Probabilistic Snap-Stabilization

This section deals with results given in [AD17]. In that paper, we have introduced a weak variant of snap-stabilization called *probabilistic snap-stabilization*.

4.2.1 Motivation and Definition

Most of classical problems have no deterministic self-stabilizing solutions in case of process-anonymity, *e.g.*, leader election in trees [YK96] or token passing in directed rings [Her90]. Probabilistic self-stabilization [IJ90, Her90], for example, have been introduced to circumvent these impossibilities. Recall that probabilistic self-stabilization differs from deterministic self-stabilization by the convergence property, *i.e.*, in probabilistic

self-stabilization, the convergence is not certain since it is only guaranteed with probability one, *i.e.*, almost surely. In that sense, probabilistic self-stabilization follows a *Las Vegas* approach.

Our idea has been to propose a probabilistic variant (an so weakened form) of snap-stabilization to offer stronger safety guarantees than probabilistic self-stabilization in anonymous networks. In other words, we aim at relaxing the definition of snap-stabilization without altering its strong safety guarantees to address anonymous networks. First, recall that a specification can always be expressed as the conjunction of a safety property and a liveness property [AS85, MP90]. Hence, we can define *probabilistic snap-stabilization* as follows: starting from an arbitrary configuration (or equivalently, after the last transient fault), a probabilistically snap-stabilizing algorithm satisfies its safety property *immediately*; whereas its liveness property is only ensured with probability 1 (*i.e.*, almost surely). Hence, compared to its deterministic counter part, the safety guarantee remains unchanged, but, we allow the algorithm to compute for a possibly long, yet almost surely finite, time. Consider for example, a dynamic specification. If an algorithm is probabilistically snap-stabilizing for that specification, then the result obtained from any initialized computation will be correct. However, such a computation both starts and terminates in almost surely finite time only.

4.2.2 Comparison with Related Properties

First, by definition, *liveness* implies *almost surely liveness*, so probabilistic snap-stabilization is weaker than its deterministic counterpart. By definition again, probabilistic snap-stabilization offers a stronger safety than deterministic self-stabilization. Indeed, safety is never violated in probabilistic snap-stabilization, while safety is only eventually satisfied in deterministic self-stabilization. However, probabilistic snap-stabilization is weaker in terms of liveness guarantees. Indeed, liveness properties are deterministically satisfied in deterministic self-stabilization. So, these two properties are not comparable. Surprisingly, probabilistic snap-stabilization and probabilistic self-stabilization are also not comparable for the following reasons. First, by definition, probabilistic snap-stabilization offers stronger safety properties than probabilistic self-stabilization. Focus now on liveness. Consider any distributed algorithm $\mathcal{A}$. Assume that $\mathcal{A}$ is probabilistically self-stabilizing w.r.t. some specification *SP*. Then, consider any legitimate configuration γ of $\mathcal{A}$. Every execution starting from γ (deterministically) satisfies *SP* (*n.b.*, only the convergence property of probabilistic self-stabilization is probabilistic). This means in particular that every execution starting from γ (deterministically) satisfies the liveness of *SP*. Assume now that $\mathcal{A}$ is probabilistically snap-stabilizing w.r.t. *SP*. Then, almost surely liveness only states that liveness of *SP* is ensured with probability 1 in all executions. This is, in particular, true for all executions starting from γ. Hence, probabilistic snap- and self- stabilization are not comparable in the general case. However, in many cases (like for the two solutions we proposed in [AD17]), an algorithm can be both probabilistically self- and snap- stabilizing.

4.2.3 Algorithmic Contribution in a Nutshell

To illustrate the property of probabilistic snap-stabilization, we have proposed two algorithms. These algorithms, in particular, show that probabilistic snap-stabilization is more expressive than its deterministic counterpart. Indeed, we propose two probabilistic snap-stabilizing algorithms for a problem having no deterministic snap- or self-stabilizing solution: the *guaranteed service leader election* in arbitrary connected anonymous networks. This problem is already defined in Section 2.4.1. Recall that the *guaranteed service leader election* problem consists in computing a correct answer to each process that, upon a request, initiates the question "Am I the leader of the network?," *i.e.*,

1. processes always compute the same answer to that question and

2. exactly one process computes the answer *true*.

Our solutions being probabilistically snap-stabilizing, the answers will be delivered within an almost surely finite time only; however any delivered answer is correct, regardless the arbitrary initial configuration and provided the question has been properly started.

Our two algorithms are designed in the atomic-state model. The first solution assumes a synchronous daemon. The second assumes a distributed unfair daemon, the most general daemon of the model. Both algorithms need an additional assumption: the knowledge of a bound B such that $B < n \leq 2B$, where n is the number of processes. The memory requirement of both algorithms is in $O(\log n)$ bits per process. Using our synchronous algorithm both the expected delay to initiate a question after a request and the expected time to obtain the associated answer are in $O(n)$ rounds, while these times are $O(n^2)$ rounds for asynchronous one. Moreover, if we add the assumption that processes know an upper bound, D, on the diameter of the network, the expected time complexity of the synchronous (resp. asynchronous) algorithm can be reduced $O(D)$ rounds (resp. $O(D.n)$ rounds); see [AD17] for detail.

4.2.4 Overview of our Algorithmic Solutions

To finally obtain a solution working under the distributed unfair daemon, we have followed a step-by-step approach:

1. First, we have designed a probabilistic non-stabilizing guaranteed service leader election working under the synchronous daemon.

2. Then, we have modified it to obtain a probabilistic snap-stabilizing solution working under the synchronous daemon.

3. Finally, we have adapted this latter algorithm to handle the asynchronous context.

As a preliminary result, we have shown that without additional global information on the network (*e.g.*, a bound on its size or its diameter), our problem cannot be solved, even in a non-stabilizing manner. We have obtained this result by reduction to the size-count specification in anonymous rings, this latter being already known to have no Las Vegas probabilistic solution; see [Tel01]. Thus, following the approach proposed in [MA89], we assume that a bound B on the number of processes n satisfying $B < n \leq 2B$ is known by all processes.

4.2.4.1 A Non-stabilizing Probabilistic Solution for Synchronous Systems

Each process p maintains a Boolean variable, $p.\ell$, which is initialized randomly; p is said to be *candidate* if and only if $p.\ell = true$. Then, our algorithm executes infinitely many cycles. Each cycle aims at computing whether there is a unique candidate. The result of this computation is stored in the variable $p.Unique$ at each process p. At the end of each cycle, all $Unique$ variables have the same value and this value is true if and only if there is a unique candidate, say q. In that case, q is designed as the leader forever, so all variables ℓ remain unchanged in all subsequent cycles. Otherwise, all variables ℓ are reset randomly.

Now, when a process p initiates a question "Am I the leader of the network?," it simply waits for the end of the current cycle: if $p.Unique$ is true, then its answer is $p.\ell$; otherwise it delays its answer at least up to the end of the next cycle. Hence, our goal is to ensure that there will exist a unique candidate in almost surely finite time.

We now explain how cycles are executed. Initially, $p.Unique$ is set to false and $p.\ell$ is set to a random value. Then, each cycle is divided into three phases. To perform a computation involving all processes, each phase should be made of at least $\mathcal{D}$ steps, where $\mathcal{D}$ is the network diameter. Thus, using the initial knowledge of processes, the length of each phase has been set to $2B > \mathcal{D}$ steps. So, each cycle contains $6B$ steps and is executed in $O(n)$ rounds. Finally, each process p maintains a local clock $p.c \in \{0,...,6B-1\}$ initialized to 0 and incremented modulo $6B$ at each step. In particular, this clock allows each process to know which phase of the cycle is running.

We now detail each phases of a cycle. The first phase consists in computing a spanning forest where a process is the root if and only if it is a candidate (of course, if there is no candidate, no tree exists at the end of the phase). During the second phase each candidate p (if any) computes in $p.cpt$ the number of processes in its tree. At the end of the phase, if a process p is candidate and $p.cpt > B$, then this means that a majority of processes belong to its tree. In this case, it sets $p.Unique$ to true. In all other cases, a process p sets both $p.\ell$ and $p.Unique$ to false. The majority being unique, at most one process is still candidate at the end of this phase. The last phase allows to broadcast the result of the two previous phases to all processes: at each step, each process p recomputes $p.Unique$ as the disjunction of $p.Unique$ and the $Unique$ variables of all its neighbors. Thus, at the end of the cycle, for every process p, $p.Unique$ is true if and only if there is a unique candidate. If a unique candidate is elected, all variables ℓ become constant and $Unique$ is always equal to true at the end of each subsequent cycles. Otherwise, all ℓ variables are randomly reset at the beginning of the next cycle.

The time complexity of our algorithm depends on the probability law used when a process assigns a Boolean value to its ℓ variable. We have computed that the best choice (considering the knowledges of processes) is that each process p assigns true to $p.\ell$ with probability $Pr \in [\frac{1}{2B}, \frac{1}{B+1}]$. In this case, the expected number of cycles to obtain a unique leader is around $\frac{e^2}{2} \leq 3,70$. Thus, the expected time to compute an answer to any question is in $O(n)$ rounds.

4.2.4.2 A Probabilistic Snap-stabilizing Solution for Synchronous Systems

If the previous algorithm starts from an arbitrary configuration, the main problem is that local clocks may be desynchronized. To resynchronize these clocks, we maintain them using a self-stabilizing *synchronous unison* algorithm which stabilizes to a legitimate configuration from which all clocks increment (modulo some period) at each step while staying equal. To that goal, we use the algorithm of Boulinier *et al.* [BPV04]. Using this algorithm, all clocks become equal after at most $6B$ steps ($6B = O(n)$). So, each cycle started after these $6B$ steps correctly computes the variable $Unique$.

Hence, to obtain the probabilistic snap-stabilization, we simply proceed as follows: when a process p initiates a question, it first waits $6B$ steps and only consider cycles started after this waiting time. Hence, it will only consider outputs computed by correct cycles. Notice also that, with this delay, the expected time to answer a question remains in $O(n)$ rounds.

4.2.4.3 A Probabilistic Snap-stabilizing Solution for Asynchronous Systems

In asynchronous systems, the unison of Boulinier *et al.* [BPV04] is self-stabilizing for a weak version of the synchronous unison problem: the *asynchronous unison*. Recall that, from any legitimate configuration, a self-stabilizing solution to this problem ensures that clocks regularly increment (modulo some period) and clocks of every two neighbors always differ by at most one increment. We use this property to emulate the computation of synchronous cycles. At each incrementation, a process saves its previous state, moreover each of its neighbors has either the same clock value, or is one increment ahead. In this latter case, the process

consider the previous state of its neighbor instead of its current one to compute its next state. Using this technique, the asymptotic space complexity is kept unchanged.

Then, the principle is similar to the synchronous algorithm, when a process initiate a question, it waits for some time before considering the output computed by a cycle. Actually, to detect when cycles are stabilized, we use a result from [BLP08]:

If $p.c$ successively takes values $u, u+1, ..., u+(2\mathcal{D}+1)$ between configurations γ_{t_0} and $\gamma_{t_{2\mathcal{D}+1}}$ with $\forall i \in \{1, ..., 2\mathcal{D}+1\}$, $u+i > 0$, then every other process executes at least one step between configurations γ_{t_0} and $\gamma_{t_{2\mathcal{D}+1}}$.

This result provides a mechanism allowing a process to locally observe whether at least one round has elapsed. Indeed, by definition, if a process observes that all processes execute at least one step, then at least one round has elapsed. So, to decide that the clock are stabilized, a process p should observe that (i) all processes execute at least X rounds, where X is any upper bound on the stabilization time of the asynchronous unison algorithm. The remainder of the implementation is similar to the synchronous case, yet tedious. So, refers to [AD17] for more detail. Just notice that the drawback of this technique is that the expected delay to obtain an answer becomes $O(n^2)$ rounds.

4.2.5 A Monte Carlo Approach

We have defined the probabilistic snap-stabilization following the Las Vegas approach in order to keep the strong safety property of the deterministic snap-stabilization. However, we can define another probabilistic variant of the snap-stabilization, following the *Monte Carlo* approach. This variant, called here the *Monte Carlo snap-stabilization*, consists in (deterministically) satisfying the liveness of the specification, while only guarantee the safety with positive (typically high) probability $1-\varepsilon$ with $\varepsilon \in (0,1)$.

Notice that, probabilistic (Las Vegas) snap-stabilization and Monte Carlo snap-stabilization are not comparable. Moreover, Monte Carlo snap-stabilization is weaker than (deterministic) snap-stabilization and not comparable to both probabilistic and deterministic self-stabilization (the reasoning is similar to the one used in subsection 4.2.2).

Our two algorithms can be easily modified to become Monte Carlo snap-stabilizing using the so-called *amplification* method [GSB94]. Recall that in those algorithm, each question is delayed enough in order to only consider results of cycles fully executed after the stabilization of the clocks. Here, we call such cycles the *safe cycles*. Moreover, the answer ($p.\ell$) to a question of some process p is delivered only once a safe cycle terminates and outputs that a unique leader exists ($p.Unique$).

The idea here consists in bounding the maximum number of safe cycles before delivering an answer to the question of p. Let X be a positive integer. If $p.Unique = false$ at the completion of all the X first safe cycles that follow the question, then p decides to blindly deliver $p.\ell$ at the end of the next cycle, no matter the value of $p.Unique$. In all other cases, p delivers $p.\ell$ at the completion of the first safe cycle that follows. Hence, at most $X+1$ safe cycles are executed: the liveness is deterministic.

Let now consider the safety part of the specification. Figure 4.1 gives the probability $1-\varepsilon$ according to the value of X. In particular, the figure shows that setting X to a constant value is enough to obtain safety with high probability. As a matter of fact, for $X = 7$, safety is achieved with probability at least 91.99 %. Note also that choosing X as a constant, we obtain time bounds of the same order of magnitude as the expected time bounds we have given for our Las Vegas algorithms.

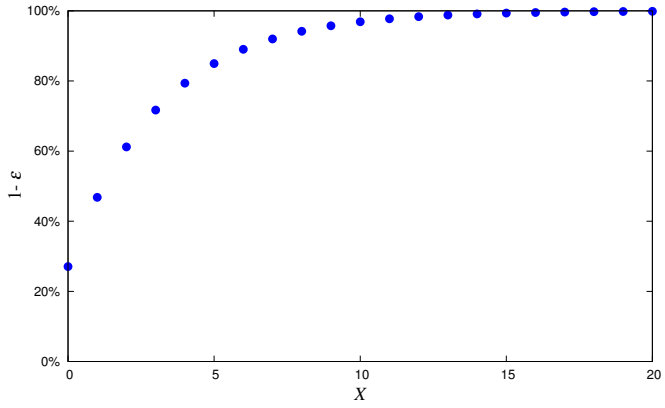


Figure 4.1: $1-\varepsilon$ versus X.

4.2.6 Expressiveness

We can easily modify our Las Vegas and Monte Carlo solutions to obtain a guaranteed service algorithm whose result is the guarantee that the whole network has been identified, *i.e.* after the termination of an initiated computation, the processes are named with constant unique identifiers.

Then, using such an algorithm, we can mimic the behavior of an identified network and emulate the transformer proposed in Chapter 2. As a consequence, every (non-stabilizing) algorithm that snap-stabilized using the transformer presented in Chapter 2 can also be turned into a probabilistic snap-stabilizing algorithm working in anonymous networks. Overall, this means that, in the atomic-state model, probabilistic snap-stabilization in anonymous networks is as expressive as deterministic self- and snap- stabilization in identified networks.

4.3 Gradual Stabilization

This section deals with results proposed in [ADDP19b]. In this paper, we have introduced a weak variant of superstabilization [DH95], yet strong variant of self-stabilization, called *gradual stabilization*.

4.3.1 Context and Definition

Up to now, self-stabilizing algorithms are mainly dedicated to static networks. Now, as explained in Section 1.3.6, those handling arbitrary topologies tolerate, up to a certain extent, some topological changes (*i.e.*, the addition or the removal of communication links or nodes). Precisely, if topological changes are eventually detected locally at involved processes and if the frequency of such events is low enough, then they can be treated as transient faults. Now, for most of the specifications (*e.g.*, those of synchronization problems [AKM$^+$93]), the stabilization time is significant, *i.e.*, $\Omega(\mathcal{D})$ rounds, where $\mathcal{D}$ is the network diameter. Hence, some works focus on proposing self-stabilizing algorithms that additionally ensure drastically smaller convergence times in favorable cases. In particular, *superstabilization* [DH95] aims at efficiently withstanding topological changes when they are sparse, and so transient. However, until now there exist only few superstabilizing algorithms for infinite problems, and those algorithms are dedicated to particular topologies, *e.g.*, token-based mutual exclusion in rings [Her00]. The apparent seldomness of superstabilizing

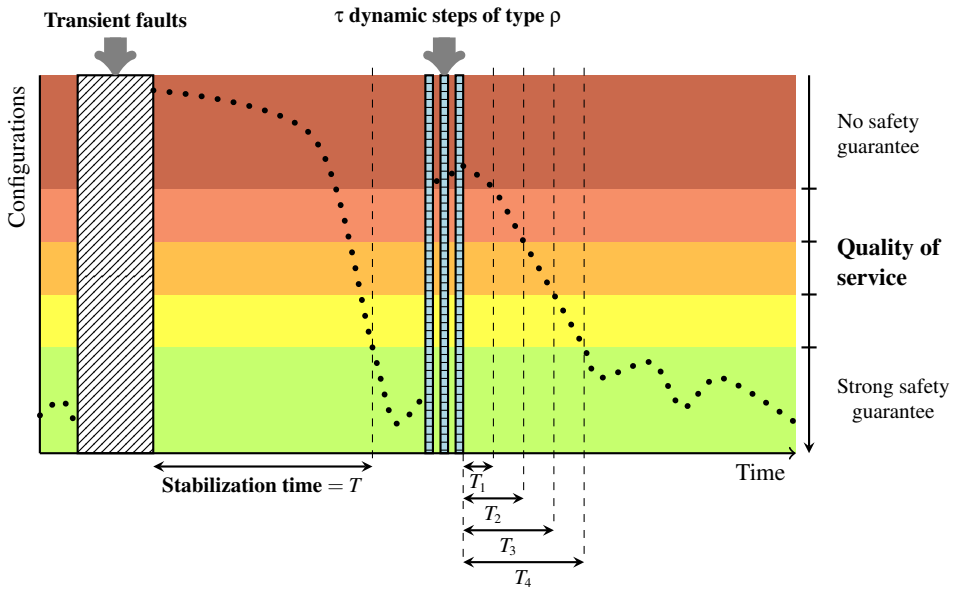


Figure 4.2: Gradual Stabilization

solutions for infinite problems, such as token-passing, may suggest the difficulty of obtaining such a strong property. We have then proposed in [ADDP19b] another strong form of self-stabilization, called *gradual stabilization under* (τ,ρ)*-dynamics*. This property also captures the ability of a self-stabilizing algorithm to efficiently withstand transient topological changes. An algorithm is gradually stabilizing under (τ,ρ)-dynamics if it is self-stabilizing and satisfies the following additional feature; see Figure 4.2 for an illustration. After up to τ *dynamic steps*[1] of type ρ[2] occur starting from a legitimate configuration, a gradually stabilizing algorithm first quickly recovers a configuration from which a specification offering a minimum quality of service is satisfied. It then gradually converges to specifications offering stronger and stronger safety guarantees until reaching a configuration (1) from which its initial (strong) specification is satisfied again, and (2) where it is ready to achieve gradual convergence again in case of up to τ new dynamic steps of type ρ. Of course, the gradual stabilization makes sense only if the convergence to every intermediate weaker specification is fast. Finally, a gradually stabilizing algorithm being also self-stabilizing, we should remark that it still recovers within finite time (yet maybe more slowly) after any other finite number of transient faults, including for example more than τ arbitrary dynamic steps or other transient failure patterns such as memory corruption.

In the following, we say that an algorithm $\mathcal{A}$ is *gradually stabilizing under* (τ,ρ)*-dynamics* for $(SP_1 \bullet T_1, SP_2 \bullet T_2, \ldots, SP_k \bullet T_k)$ if $\mathcal{A}$ is self-stabilizing for SP_k and after at most τ dynamic steps of type ρ from a legitimate configuration, $\mathcal{A}$ first stabilizes to SP_1 in at most T_1 rounds, then stabilizes to SP_2 within at most T_2 additional rounds, and so on until stabilizing again to SP_k. Of course, we should have $SP_k \Rightarrow SP_{k-1} \Rightarrow \ldots \Rightarrow SP_1$.

[1] *N.b.*, a dynamic step is a step containing topological changes.

[2] Precisely, ρ is a binary predicate over graphs, representing network topologies, such that $\rho(G,G')$ is true if and only if it is possible for the system to switch from topology G to topology G' in a single (dynamic) step.

4.3.2 Comparison with Related Properties

Gradual stabilization is related to two other stronger forms of self-stabilization, namely *safely converging self-stabilization* [KM06] and *superstabilization* [DH95].

Recall that the goal of a *safely converging self-stabilizing algorithm* is first to quickly (within $O(1)$ rounds is the usual rule) converge from any arbitrary configuration to a *feasible* legitimate configuration, where a minimum quality of service is guaranteed. Once such a feasible legitimate configuration is reached, the system continues to converge to an *optimal* legitimate configuration, where more stringent conditions are required. Hence, the aim of safely converging self-stabilization is also to ensure a gradual convergence, but only for two specifications. However, this kind of gradual convergence should be ensured after any number of transient faults (such transient faults can include topological changes, but not only), while the gradual convergence of our property applies after dynamic steps from legitimate configurations only.

A *superstabilizing algorithm* is self-stabilizing and has two additional properties. In presence of a single topological change (adding or removing one link or process in the network), it recovers fast (typically $O(1)$), and a safety predicate, called a *passage* predicate, should be satisfied all along the stabilization phase. Like in the gradual stabilization approach, the fast convergence, captured by the notion of *superstabilization time*, and the passage predicate should be ensured only if the system was in a legitimate configuration before the topological change occurs. In contrast with the gradual stabilization approach, superstabilization consists in only *one* dynamic step satisfying a very restrictive dynamic pattern, noted here ρ_1: only one topological event, *i.e.*, the addition or removal of one link or process in the network. Hence, a superstabilizing algorithm for a specification SP and with the passage predicate P can be seen as an algorithm which is gradually stabilizing under $(1,\rho_1)$-dynamics for $(P \bullet 0, SP \bullet f)$ where f is its superstabilization time.

4.3.3 An Illustrative Example

4.3.3.1 Unison Variants

We have illustrated this new property by considering three variants of a classical synchronization problem respectively called here *strong*, *weak*, and *partial* unison. In these problems, each process should maintain a local clock. We restrict here our study to periodic clocks, *i.e.*, all local clocks are integer variables whose domain is $\{0,\ldots,\alpha-1\}$, where $\alpha \geq 2$ is called the *period*. Each process should regularly increment its clock modulo α (liveness) while fulfilling some safety requirements. The safety of strong unison imposes that *at most two consecutive clock values* exist in any configuration of the system. Weak unison —also called *asynchronous unison* in the literature— only requires that the difference between clocks of every two neighbors is at most one increment. Finally, we have defined partial unison as a property dedicated to dynamic systems, which only enforces the difference between clocks of neighboring processes present before the dynamic steps to remain at most one increment.

4.3.3.2 A Gradually Stabilizing Unison Algorithm

Assuming anonymous processes, we have proposed an algorithm written in the atomic-state model that gradually stabilizes to these three specifications, respectively denoted by SP_{PU} (partial unison), SP_{WU} (weak unison), and SP_{SU} (strong unison), under the distributed unfair daemon. Precisely, our algorithm is gradually stabilizing under $(1,\mathsf{BULCC})$-dynamics for $(SP_{\mathsf{PU}} \bullet 0, SP_{\mathsf{WU}} \bullet 1, SP_{\mathsf{SU}} \bullet (\mu+1)\mathcal{D}_1+2)$ where $\mathcal{D}_1$ is the diameter of the network after the dynamic step and μ is a parameter satisfying $\mu \geq \max(2,N)$, where N is a given upper bound on the number of processes existing in any reachable configuration. The condition BULCC restricts the gradual convergence obligation to dynamic steps, called BULCC-dynamic steps, that fulfill the following

conditions. A BULCC-dynamic step may contain several topological events, *i.e.*, link and/or process additions and/or removals. However, after such a step, the network should (1) contain at most N processes, (2) stay connected, and (3) if $\alpha > 3$, every process which joins the system should be linked to at least one process already in the system before the dynamic step, unless all of those have left the system. Condition (1) is necessary to have finite periodic clocks. We have shown the necessity of condition (2) to obtain our results whatever the period is, while we have proven that condition (3) is necessary for our purposes when the period α is fixed to a value greater than 5. Finally, we have exhibited pathological cases for periods 4 and 5, in case we do not assume condition (3).

Our algorithm is gradually stabilizing because after one BULCC-dynamic step from a configuration which is legitimate for the strong unison, it immediately satisfies the specification of partial unison, then converges to the specification of weak unison in at most one round, and finally retrieves, after at most $(\mu+1)\mathcal{D}_1+1$ additional rounds (where $\mathcal{D}_1$ is the diameter of the network after the dynamic step), a configuration (i) from which the specification of strong unison is satisfied, and (ii) where it is ready to achieve gradual convergence again in case of another dynamic step.

Again the algorithm being also self-stabilizing (by definition), it still converges to a legitimate configuration of the strong unison after the system suffers from arbitrary transient faults including, for example, several arbitrary dynamic steps. However, in such cases, there is no safety guarantee during the stabilization phase.

4.3.3.3 Overview of our Solution

Our solution is inspired from the asynchronous unison algorithm proposed by Boulinier in his thesis. This latter is actually a generalization of the asynchronous unison proposed by Couvreur *et al.* [CFG92]. Our algorithm uses three parameters: α, μ and β. These parameters should satisfy the following conditions: $\alpha \geq 2$, $\mu \geq \max(2,N)$ (where N is a given bound on the number of processes existing in any reachable configuration), and $\beta = K.\alpha$ with $\beta > \mu^2$ and $K > N$. Each process p maintains two local clocks at each process p: an *external* clock $p.c \in \{0,...,\alpha-1\} \cup \{\perp\}$ and an *internal* clock $p.t \in \{0,...,\beta-1\}$.

Self-stabilizing strong unison. Starting from an arbitrary configuration, the algorithm makes the internal clocks converging so that they eventually satisfy the weak unison specification. The values of external clocks are then computed from the internal clocks to obtain a strong unison. To that goal, we use the notion of *delay* : the delay $d_\beta(x,y)$ between two integer values x and y is equal to $\min((x-y) \bmod \beta, (y-x) \bmod \beta)$. We also use the relation $\preceq_{\beta,\mu}$ between two integer values x and y defined as follows: $x \preceq_{\beta,\mu} y \equiv ((y-x) \bmod \beta) \leq \mu$.

When a process p detects that the weak unison is locally correct, *i.e.*, when the delay between its internal clock and those of each of its neighbors is at most 1, p can increment its internal clock $p.t$ when $p.t$ is late or at the same time compared to each neighboring internal clock; here when the delay between $p.t$ and $q.t$ is at most one, for every neighbor q.

Conversely, when p locally detects that the weak unison is not satisfied, we have two cases: either the delay between p and each of its neighbors is at most μ and, in this case, p behaves as previously, or its internal clock is too late compared to a neighboring internal clock and then $p.t$ is reset to 0 (of course, only if $p.t \neq 0$). This mechanism allows the system to stabilize to a configuration where the internal clocks satisfy the weak unison specification.

Then, to allow the external clocks to stabilize to the strong unison, the value of an external clock is simply recomputed each time the associated internal clock is modified. Precisely, the value of $p.c$ is maintained equal to $\lfloor \frac{\alpha}{\beta} p.t \rfloor$: $p.c$ is computed from $p.t$ as a normalization operation from clock values in $\{0,\ldots,\beta-1\}$ to $\{0,\ldots,\alpha-1\}$. We set β in such a way that $K = \frac{\beta}{\alpha}$ is greater than N to ensure that, when the delay between

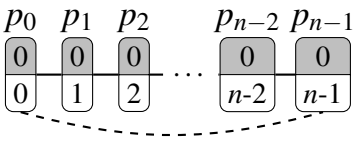


Figure 4.3: Link addition.

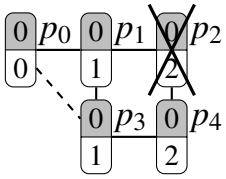


Figure 4.4: Node removal.

any two t-clocks is at most $N-1$, the delay between any two c-clocks is at most one. Furthermore, the liveness of the weak unison ensures that every t-clock increments infinitely often, hence so do c-clocks.

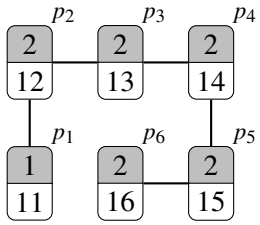
Our algorithm is in a legitimate configuration (w.r.t. the strong unison) when each process p satisfies $p.c \neq \perp$ and $p.c = \lfloor \frac{\alpha}{\beta} p.t \rfloor$ and $d_\beta(p.t, q.t) \leq 1$, for each neighbor q (the necessity of $\perp$ will be justified later). Under these conditions, the delay between internal clocks is bounded by $N-1$, so the delay between external clocks of any two processes is at most 1: the external clocks satisfy the strong unison specification.

We now illustrate the gradual stabilization property under $(1, \mathsf{BULCC})$-dynamics of our algorithm with several scenarios. Precisely, we consider several kinds of possible dynamic steps $\gamma_i \mapsto \gamma_{i+1}$ satisfying the condition BULCC.

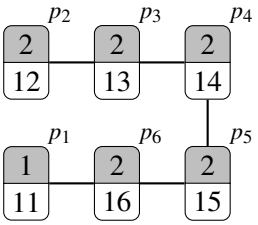
Link additions. Assume first that $\gamma_i \mapsto \gamma_{i+1}$ contains link additions only. Adding a link (see the dashed link in Figure 4.3) can break the safety of weak unison on internal clocks. Indeed, it may create a delay greater than one between two new neighboring t-clocks. Nevertheless, the delay between any two t-clocks remains bounded by $n-1 \leq N-1$, where n is the actual number of processes in γ_{i+1}. Consequently, no process will reset its t-clock (Figure 4.3 shows a worst case). Moreover, c-clocks still satisfy strong unison specification immediately after the link addition. Besides, since increments are constrained by neighboring clocks, adding links only reinforces those constraints. Thus, the delay between internal clocks of arbitrary far processes remains bounded by $n-1 \leq N-1$, and so strong unison remains satisfied, in all subsequent static steps. Consider again the example in Figure 4.3: before $\gamma_i \mapsto \gamma_{i+1}$, p_{n-1} had only to wait until p_{n-2} increments $p_{n-2}.t$ in order to be able to increment its own t-clock; yet after the step, it also has to wait for p_0 until its internal clock reaches at least $n-1$.

Process and link removals. Assume now that $\gamma_i \mapsto \gamma_{i+1}$ contains process and link removals only. By definition of BULCC, the network remains connected. Hence, constraints between (still existing) neighbors are maintained: the delay between t-clocks of two neighbors remains bounded by one; see the example in Figure 4.4: process p_2 and link $\{p_0, p_3\}$ are removed. So, weak unison on t-clocks remains satisfied and so is strong unison on c-clocks.

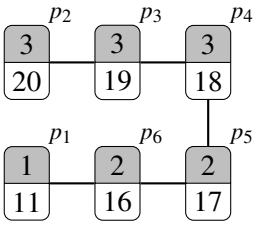
Link additions coupled with both process and link removals. Consider now a more complex scenario, where $\gamma_i \mapsto \gamma_{i+1}$ contains link additions as well as process and/or link removals. Figure 4.5 shows an example



(a) Initial configuration that is legitimate for the strong unison.



(b) After one dynamic step: link $\{p_1, p_2\}$ disappears and link $\{p_1, p_6\}$ is created.



(c) Some steps later, strong unison is violated.

Figure 4.5: Example of execution where one link is added and another is removed: $\mu = 6$, $\alpha = 7$, and $\beta = 42$.

of such a scenario, where safety of strong unison is violated. As above, the addition of link $\{p_1, p_6\}$ in Figure 4.5b leads to a delay between t-clocks of these two (new) neighbors which is greater than one (here 5). Moreover, the removal of link $\{p_1, p_2\}$, also in Figure 4.5b, relaxes the neighborhood constraint on p_2: p_2 can now increment without waiting for p_1. Consequently, after several static steps from Figure 4.5b, the system can reach Figure 4.5c, where the delay between p_1 and p_2 is 9. Due to the fact that c-clock values are computed from t-clock values, the strong unison safety is also violated: the system reaches the configuration presented Figure 4.5c where we have three different values in c-clocks: $p_1.c = 1$, $p_6.c = 2$, and $p_2.c = 3$. In the worst case scenario, after $\gamma_i \mapsto \gamma_{i+1}$, the delay between two neighboring t-clocks is bounded by $n-1 \leq N-1$, so (1) the maximum delay between neighbors remains bounded by $N-1$ in the following steps since no internal clock is reset and (2) the system converges to a configuration where this maximum delay is at most 1: the external clocks violate the strong unison safety, but continue to satisfy the weak unison safety while converging to the strong unison again.

Process additions. Finally, assume that some processes join the system: in this case, each new process p sets $p.t$ to 0 and $p.c$ to $\bot$. Because of our assumption on the dynamic steps, the partial unison is immediately satisfied. Then, to ensure that the weak unison is obtained within the next round, each new process p is enabled to assign a value to $p.c$. Because of the condition BULCC, p has at least one neighbor q such that $q.c \neq \bot$. The clock $p.t$ is then set to the minimum value held by a neighbor q satisfying $q.c \neq \bot$ and $p.c$ is set according to the value taken by $p.t$ as explained before.

Notice that to ensure the self-stabilization of our algorithm, we should prevent deadlock in case of arbitrary initialization where, for example, the internal clocks of all processes are equal to $\bot$: a process p both sets $p.t$ and $p.c$ to 0 when p and all its neighbors have their internal clock equal to $\bot$.

4.3.4 Possible Extensions

Concerning our unison algorithm, the graceful recovery after one dynamic step comes at the price of slowing down the clock increments. The question of limiting this drawback remains open. Furthermore, it would be interesting to address in future work gradual stabilization for other problems (in particular infinite one) in context of more complex dynamic patterns.

Chapter 5

Unifying Versatility and Efficiency in Self-Stabilization

In this chapter, we review several works [CDV09a, DJ19, DIJ19] where we could unify both versatility and efficiency by proposing relatively general algorithmic schemes whose instantiations are efficient. All those works have been achieved assuming the atomic-state model under the distributed unfair daemon.

5.1 General Self-Stabilizing Scheme for Tree-based Constructions

In distributed computing, many problems are closely related, and even sometime computationally equivalent. By computationally equivalent, we mean that there exists a two-way reduction: the former problem can be solved in a system where the latter is solvable, and conversely. For example, leader election and spanning tree construction are computationally equivalent [San06]. As a matter of facts, all existing self-stabilizing leader election algorithms for identified connected networks actually also build a spanning tree rooted at the elected process; see, *e.g.*, [ACD$^+$17b]. Similarly, most of existing self-stabilizing k-clustering algorithms are based on a spanning tree construction; see [DLD$^+$13, DDH$^+$16].

In [DIJ19, DIJ20],[1] we exploit the many common features shared by all these problems to propose a general algorithmic scheme that provides efficient silent tree-based data structure constructions.

To be as general as possible, our scheme is based on several inputs and assumptions, listed below.

5.1.1 Inputs and Assumptions

5.1.1.1 Networks

We consider bidirectional networks which are not necessarily connected. When the network is not connected, we may have connected components in which we do not want to build trees, due to the problem specification. In this case, we enforce all processes of such components to converge to a special terminating state. This allows us to implement what we have called "non-rooted components detection" in [DIJ17], in which case the special state notify the process about the absence of any root in its connected component.

We need weights to express problems such as the shortest path (spanning) tree. To that goal, we assume that each undirected edge $\{p,q\}$ actually consists of two arcs: (p,q) (*i.e.*, the directed link from p to q) and

[1] The conference paper [DIJ19] was a preliminary version. Many results we present here have been added after and so are present in the journal version [DIJ20] which is under submission.

(q,p) (*i.e.*, the directed link from q to p). Each arc (p,q) has a weight, denoted by $\omega_p(q)$. Notice that we may have $\omega_p(q) \neq \omega_q(p)$. Weights belong to a given domain *DistSet*, whose properties will be presented later.

5.1.1.2 Inputs

Below, we list and explain the three main inputs used by each process p in our algorithmic scheme. (Notice that our scheme also requires few problem dependent predicates that we will not present here; see [DIJ20] for more details.)

$pname_p$**:** the name of p.

$pname_p \in IDs$, where $IDs = \mathbb{N} \cup \{\perp\}$ is totally ordered by $<$ and $\min_<(IDs) = \perp$. The value of $pname_p$ is problem dependent. Actually, we consider two particular cases of naming. In one case, $\forall q \in V, pname_q = \perp$. In the other case, $\forall p,q \in V, pname_p \neq \perp \wedge (p \neq q \Rightarrow pname_p \neq pname_q)$, *i.e.*, $pname_p$ is a unique global *identifier*.

$canBeRoot_p$**:** a Boolean input.

$canBeRoot_p$ is true if p is allowed to be the root of a tree. In this case, p is called a *candidate*. In a terminal configuration, every tree root satisfies *canBeRoot*, but the converse is not necessarily true (it depends on the problem we consider).

Moreover, for every connected component $\mathcal{C}$, if there is at least one candidate $p \in \mathcal{C}$, then every process of $\mathcal{C}$ should belong to a tree (so there is at least one tree root in $\mathcal{C}$) in any terminal configuration.

If there is no candidate in a connected component, we require that all processes of the component converge to a particular terminal state as explained before.

Actually, together with *pname*, *canBeRoot* allows us to express various kinds of situations. For example, in a rooted semi-anonymous system, *canBeRoot* is true at exactly one process and all processes satisfy $pname = \perp$.

When considering the leader election in identified networks, *canBeRoot* is true at all processes (meaning that all processes compete in the election) and *pname* gives the unique identifier at each process.

We can also consider other semi-anonymous systems, where we have, for example, several roots (like in the disjunction problem [DDL17]), in which case *canBeRoot* is true at each root and all processes satisfy $pname = \perp$.

Finally, decoupling the labeling from the fact of being root (or candidate) is interesting to express some problems such as the *generalized minimal* k*-dominating set problem* [DDL19]. Indeed, in that problem, we want to build, in an identified network, a minimal k-dominating set which is a superset of a given set I of processes. This problem can be instantiated in our scheme by defining I as a set of candidates.

$distRoot_p$**:** a constant belonging to *DistSet*.

Every tree is based on some kind of distance, *e.g.*, the number of hops to the root. Thus, $distRoot_p$ is the distance value of process p if p is finally a root of some tree. For example, $distRoot_p = 0$ for the BFS spanning tree construction: if p is the unique root, *i.e.*, $canBeRoot_p = true$, then its distance variable should be equal to $distRoot_p$, *i.e.*, 0.

Notice that the distances are not always integers. For example, in our instantiation of the DFS spanning tree (which is inspired from the Collin and Dolev' algorithm [CD94]), the distance of process p is a string corresponding to a list of incoming channel numbers in the tree path from the root to p. In this case, $distRoot_p$ is a minimum value in the lexicographical order used to compare those strings.

5.1.1.3 The Distance Domain

We use distances in the algorithm mainly to detect cycles. Moreover, according to the specific problem we consider, we may want to minimize the weight of the trees using the distances.

Distances are computed using the arc weights. Recall that those weights belong to the domain *DistSet*. More precisely, we need an ordered magma $(DistSet, \oplus, \prec)$, *i.e.*, $\oplus$ is a closed binary operation on *DistSet* and $\prec$ is a total order on this set. The definition of $(DistSet, \oplus, \prec)$ is problem dependent. For example, for the shortest path tree, $(DistSet, \oplus, \prec)$ may be $(\mathbb{N}^*, +, <)$. For the DFS spanning tree $(DistSet, \oplus, \prec)$ may be $(\{0, \ldots, \Delta\}^*, \cdot, \prec_{lex})$, *i.e.*, assuming channels are labeled using numbers, $\{0, \ldots, \Delta\}^*$ is the infinite set of finite words made with those labels, "$\cdot$" is the concatenation operator, and $\prec_{lex}$ is the lexicographical order.

We assume that, for every edge $\{p, q\}$ of E and for every value d of *DistSet*, we have $d \prec d \oplus \omega_p(q)$ and $d \prec d \oplus \omega_q(p)$. Finally, notice that *DistSet* may or may not be an infinite set. When the cardinal of *DistSet* is finite, we obtain an instantiation with a bounded memory requirement.

5.1.2 Overview of the General Scheme

Our scheme provides silent solutions. However, we could re-use many techniques applied in the PIF algorithm presented Section 2.3, which is though a wave protocol.

More precisely, each process maintains, again, a status and a distance variable, as well as, a parent pointer. Clearly, the memory requirement of our solution highly depends on the domain of the distance variable, *i.e.*, *DistSet*, which is problem dependent. Again, we obtain a silent solution using a bounded memory per process if and only if *DistSet* is a finite set.

We use these variables to, in particular, detect inconsistencies locally at some processes, henceforth called abnormal processes. Each abnormal process together with its normal descendents shape an abnormal tree, similarly to Section 2.3. Abnormal trees are removed by first performing a freezing PIF on them, and then they are cleaned top-down.

Constructions of normal trees (maybe only one) are initiated from the candidates. Then, if the specification requires optimizations, trees may merge: a root may lose its status by becoming a child of some neighbors to minimize its distance. Moreover, the structure of the trees are locally adjusted on the fly when necessary: a process may change its parent in the tree to minimize its distance.

5.1.3 Complexity

Despite its versatility, our scheme is efficient. Indeed, its stabilization time is at most $4n_{\texttt{maxCC}}$ rounds, where $n_{\texttt{maxCC}}$ is the maximum number of processes in a connected component.

Moreover, its stabilization time in moves is polynomial in the usual cases; see the example instantiations below. Precisely, we exhibit polynomial upper bounds on its stabilization time in moves that depend on the particular problems we consider.

Besides, most of these instantiations can be set to require only bounded memory at the price of providing to processes some knowledge about the graph topology, typically an upper bound on the number of processes. Moreover, notice that there is no overhead in moves in our instantiations when enforcing the memory to be bounded.

5.1.4 Instantiations

To illustrate the versatility and efficiency of our approach, we have proposed several of its possible instantiations for solving classical spanning-tree-based problems. One can easily derive from these various examples

many other silent self-stabilizing spanning-tree-based constructions.

Spanning forest. Assuming an input set of roots, we propose an instantiation to compute a spanning forest of arbitrary-shaped trees, with non-rooted components detection. This instantiation stabilizes in $O(n_{\texttt{maxCC}} \cdot n)$ moves, which matches the best known move complexity for spanning tree construction [Cou09a] with explicit parent pointers.

Actually, there exists a solution with implicit parent pointer [KK05] that achieves a better complexity, precisely $O(n \cdot \mathcal{D})$ moves, where $\mathcal{D}$ is the network diameter. However, adding a parent pointer to this algorithm makes this solution more costly than ours in a large class of networks. Indeed, Cournier [Cou09b] has shown that the straightforward variant of this algorithm where a parent pointer variable is added has a stabilization time in $\Omega(n^2 \cdot \mathcal{D})$ steps in an infinite class of networks.

Shortest-path and DFS spanning trees. Assuming then a rooted network with positive integer weights, we propose shortest-path spanning tree and DFS constructions, with non-rooted components detection. The shortest-path spanning tree construction stabilizes in $O({n_{\texttt{maxCC}}}^3 \cdot n \cdot \texttt{W}_{\max})$ moves ($\texttt{W}_{\max}$ is the maximum weight of an edge). This move complexity matches the best known move complexity for this problem [DIJ17].

Leader election. Assuming now that the network is identified (*i.e.*, processes have distinct IDs), we propose two instantiations for electing a leader in each connected component and building a spanning tree rooted at each leader. In one version, stabilizing in $O({n_{\texttt{maxCC}}}^2 \cdot n)$ moves, the trees are of arbitrary topology. This move complexity matches the best known step complexity for leader election [ACD+17b]. In the other version, stabilizing in $O({n_{\texttt{maxCC}}}^3 \cdot n)$ moves, the leader is guaranteed to be the process of minimal identifier in the connected component, and trees are BFS.

***k*-clustering.** In the case of an identified network, we propose an instantiation solving the k-clustering problem, in which processes gather into clusters such that no process is at distance larger than k from its clusterhead. Our k-clustering solution is also a $O(k^2)$-approximation of the optimum k-clustering (in terms of number of clusters) in any Unit Disk Graph (UDG).

Optimum-bandwitdh-path (spanning) tree. Finally, assuming a rooted connected network with a bandwidth assigned to each edge, we propose an instantiation computing for each process the maximum bottleneck bandwidth to the root, and a corresponding path (with the fewest edges).

5.1.5 Related Work

As already mentioned, several mechanisms used in this general scheme have been first used to implement our PIF algorithm given in [CDD+16].

This work is also a generalization of [DIJ17], where we have also considered the atomic-state model under the distributed unfair daemon. The proposed algorithm is efficient both in terms of rounds and moves, tolerates disconnections, but is dedicated to the case of the shortest-path tree in a rooted network.

Another closely related work is the one of Cobb and Huang [CH09]. In this paper, a generic self-stabilizing algorithm is presented for constructing in a rooted connected network a spanning tree where a given metric is maximized. Now, since the network is assumed to be rooted (*i.e.*, a leader process is already known), leader election is not an instance of their generic algorithm. Similarly, since they assume connected

networks, the non-rooted component detection cannot be expressed too. Finally, their algorithm is proven in the atomic-state model, yet assuming a strong scheduling assumption: the sequential weakly-fair daemon.

5.1.6 Extensions

Many self-stabilizing silent algorithms actually consist in the composition of a spanning treelike (*e.g.* tree or forest) construction and some other algorithms specifically designed for tree/forest topologies. Those latter algorithms are usually implemented using both top-down and bottom-up actions. In [ADD18], we have formalized these design patterns to obtain general statements regarding both correctness and time complexity. Our results allow to easily (*i.e.* quasi-syntactically) deduce polynomial upper bounds on move and asymptotically optimal round complexities of algorithms based on top-down and bottom-up actions.

Hence, we can expect that composing a spanning treelike construction with such tree-based algorithms give an efficient solution. However, until now, no composition operator offers move or step complexity bounds for the output composite algorithm.

For example, the *hierarchical collateral composition* we have introduced in [DLD$^+$13] is provided only with a sufficient condition that also gives a bound of the stabilization time in rounds. Hence, a composition operator offering a simple sufficient condition together with interesting upper bounds on the stabilization time in rounds, steps, and moves still need to be sought.

5.2 Reset-Based Stabilizing Schemes

Many proposed general methods [KP93, APVD94, AG94, AO94] are based on *reset* algorithms. A reset algorithm may be centralized at a leader process, or fully distributed, meaning multi-initiator. In the former case, either the reset is coupled with a snapshot algorithm (*e.g.*, [CDD$^+$16]), or processes detecting an incoherence (using for example *local checking* [APSV91]) should request a reset to the leader. In the fully distributed case, resets are locally initiated by processes detecting inconsistencies; see *e.g.* [KNKM18].

Below, we present two reset-based methods that allow to obtain efficient stabilizing solutions in the atomic-state model assuming a distributed unfair daemon. The former uses a mono-initiator reset algorithm and allows to built snap-stabilizing wave algorithms. The latter is based on a fully distributed reset algorithm and provides time-efficient self-stabilizing algorithms.

5.2.1 Snap-Stabilizing Waves

In [CDV09a], we focus on a particular class of wave algorithms. Namely, the mono-initiator wave algorithms where decisions occur only at the initiator. Although this class seems to be restrictive, it gathers some fundamental protocols such as PIF and token circulation. Moreover, these protocols are key tools for solving a wide class of problems including reset, snapshot, mutual exclusion, routing, and synchronization.

We consider these algorithms in a rooted connected network where the root is the initiator.

5.2.1.1 Overview

The overall idea is to provide a non fault-tolerant (and so non-stabilizing) algorithm `A` yet achieving an additional property that should be easier to obtain than stabilization. This property is related to termination and so allow to combine the input algorithm with a transformer that do not use (heavy) snapshots to obtain a snap-stabilizing solution.

Starting from a predefined initial configuration γ_I, A should perform (on request) a specific wave, *i.e.*, a depth-first token circulation, that should eventually terminate at the root. Hence, at the termination, the root r is in a specific local state, say T_r. The idea is now to ensure that r eventually switches to the local state T_r even if the initial configuration is arbitrary. We then call the property satisfied by such an algorithm the *BreakingIn* property.

We showed that the *BreakingIn* property is strictly induced by self- and snap- stabilization in the class of algorithms we consider. In other words, in our context, *BreakingIn* is easier to obtain than self- or snap-stabilizing solutions. This property allows us to easily obtain snap-stabilizing solutions, indeed, upon a request, the transformer simply consists in waiting until the root gets state T_r and then snap-stabilizingly resets the system (using the reset proposed in Section 2.4.2, page 40), and finally executes A starting γ_I.

Consider for example the depth-first token circulation problem. We can easily design an algorithm satisfying *BreakingIn* for that problem. Each process has a pointer that either designates a neighbor or is in one the two following special statuses: *idle* and *done*. *idle* means that the process is not involved in the DFS circulation. Hence, the predefined initial configuration γ_I is the one where every process has status *idle*. The *done* status means that the DFS circulation from the process is now terminated. Hence, T_r is the *done* status at the root.

Starting from γ_I, it is easy to perform a DFS token circulation. The root starts by designating a neighbor. When designated by a neighbor, an idle process designates one of its *idle* neighbor, if any; otherwise it takes status *done*. Similarly, when the process points to a neighbor in the *done* status, it modifies its pointer to designates another neighbor in *idle* status, if any; otherwise it takes status *done*.

If the initial configuration is now arbitrary, we should slightly update the previous algorithm so that a process takes a *done* earlier when it detects any local inconsistency, *e.g.*, when a process is pointed by several neighbors or when a non-root process points to a neighbor, but is not pointed by another neighbor, *etc.* Such simple local corrections ensure the absence of deadlock since it enforces the root to eventually take state T_r.

5.2.1.2 Instantiations

Using this simple method, we could design several efficient snap-stabilizing algorithms.

DFS token circulation. First, the snap-stabilizing DFS token circulation we proposed perform a DFS traversal of the network in $O(n)$ rounds and $O(\Delta \times n^3)$ moves, where n is the number of processes and Δ is the maximum degree of the network. Moreover, it can be implemented using $O(\log n)$ bits per process, if processes know an upper bound of n in $O(n)$. To date, this solution is still the best snap-stabilizing one for arbitrary rooted networks.

BFS wave. Then, we provided a snap-stabilizing wave algorithm that constructs a BFS spanning tree. This latter performs a wave in $O(\mathcal{D}^2 + n)$ rounds and $O(\Delta \times n^3)$ moves bits per process ($\mathcal{D}$ is the network diameter). Moreover, it can be implemented using $O(\log n)$ bits per process, if processes know an upper bound of n in $O(n)$.

Mutual exclusion. Finally, using particular properties of our snap-stabilizing reset algorithm, we could implement a simple counting mechanism in our snap-stabilizing token circulation to locally detect the uniqueness of the token at any process. Thank to this mechanism, we could design the first snap-stabilizing mutual exclusion algorithm for arbitrary rooted and connected networks.

Beyond these examples, it would be interesting to try to generalize this approach to multi-initiator algorithms.

5.2.2 Distributed Cooperative Reset

In [DJ19], we have proposed a self-stabilizing reset algorithm working in *anonymous* networks. This algorithm resets the network in a *distributed* non-centralized manner, as each process detecting an inconsistency may initiate a reset. It is also *cooperative* [KNKM18] in the sense that it coordinates concurrent reset executions in order to gain efficiency.

Our approach is general since our reset algorithm allows to build self-stabilizing solutions for various problems and settings. As a matter of fact, it applies to both finite (*e.g.*, spanning constructions) and infinite tasks (*e.g.*, clock problems) since we propose efficient self-stabilizing reset-based algorithms for the 1-minimal (f,g)-alliance (see the detail below) in identified networks and the asynchronous unison problem in anonymous networks. These two latter instantiations enhance the state of the art. Indeed, in the former case, our solution is more general than the previous ones; while in the latter case, the time complexity of the proposed asynchronous unison algorithm is better than that of previous ones. Notice that, in the finite case, the self-stabilizing solution we obtain is also silent.

5.2.2.1 Overview

Our method is based on local checking and is fully distributed (*i.e.*, multi-initiator). It aims at reinitializing an input algorithm `I` when necessary. The reset mechanism is self-stabilizing in the sense that its composition with `I` gives a self-stabilizing solution to the specification of `I`.

Our reset mechanism works in anonymous networks and is actually multi-initiator: a process p can initiate a reset whenever it locally detects an inconsistency in `I`, *i.e.*, whenever the predicate $\neg$**P_ICorrect**(p) holds (*i.e.*, `I` is locally checkable). So, several resets may be executed concurrently. Concurrent resets have to be *cooperative* (in the sense of [KNKM18]) to ensure the fast convergence of the system to a consistent global state. More precisely, they are coordinated: a reset may be partial since we try to prevent resets from overlapping. Resets are actually *dag* (directed acyclic graph) rooted at their initiators and those dags may span the network. When a reset cannot be further propagated a convergcast of perform from the leaves to its root (the initiator) to notify it from the termination. Then, the reset dag is cleaned top-down and a process can start executing again `I` when no process in its close neighborhood (*i.e.*, the process and its neighbors) is involved in a reset. Hence, when a process restarts executing `I`, it behaves as if the configuration was already correct, *i.e.*, as if all processes were satisfying **P_ICorrect**.

As a matter of fact, our algorithm makes an input algorithm recovering a consistent global state within at most $3n$ rounds, where n is the number of processes. During recovering, any process executes at most $3n+3$ moves of the reset algorithm.

5.2.2.2 Requirements on the Input Algorithm

The input algorithm `I` should provide three inputs at each process p: the two input predicates **P_ICorrect**(p) and **P_reset**(p), and the macro *reset*(p). **P_ICorrect**(p) indicates whether p is locally correct w.r.t. `I`. **P_reset**(p) is true if and only if p is in its reset state. Finally, the macro *reset*(p) contains the assignment allowing to reset the `I` variables of p so that, after all processes have reset, the system is in a correct configuration, *i.e.*, a configuration where all processes satisfy **P_ICorrect**.

Those inputs should satisfy the following requirements:

- **P_ICorrect**(p) is closed by I, *i.e.*, whenever **P_ICorrect**(p) holds in a configuration, then **P_ICorrect**(p) still holds after any step of `I`.

- **P_reset**(p) should be defined over variables of p (in I) only.
- If $\neg$**P_ICorrect**(p) holds, *i.e.* p locally detects an inconsistency in I, then no rule of Algorithm I is enabled at p.
- If **P_reset** at all processes in the close neighborhood of p, then **P_ICorrect**(p) holds.
- If p performs a move in $\gamma \mapsto \gamma'$, where, in particular, it modifies its variables in Algorithm I by executing *reset*(p) (only), then **P_reset**(p) holds in γ'.

Notice that the **P_ICorrect** predicates can be seen as a *proof-labeling scheme* [KKP10] with a few additional features. Indeed, a proof-labeling scheme is a mechanism to certify the legitimacy of a configuration with respect to a Boolean predicate. Basically, each process p is given a local predicate $P(p)$ and the configuration is legitimate if and only if $P(p)$ holds for every process p.

Notice also that the inputs required by our reset algorithm are essentially local, *i.e.*, they depend on the close neighborhood of a process.

5.2.2.3 Instantiations

To show the efficiency of our method, we have proposed two reset-based self-stabilizing algorithms, respectively solving the asynchronous unison problem in anonymous networks and the 1-minimal (f,g)-alliance in identified networks.

Unison. We have first considered the problem of *asynchronous unison*; defined in Section 4.2.4.3, page 64. Our unison algorithm has a stabilization time in $O(n)$ rounds and $O(\Delta.n^2)$ moves, where n is the number of processes and Δ the maximum degree of the network. Actually, its stabilization time in rounds matches the one of the previous best existing solution [BPV04]. However, it achieves a better stabilization time in moves, since the algorithm in [BPV04] stabilizes in $O(\mathcal{D}.n^3 + \alpha.n^2)$ moves (as we have shown in [DP12]), where $\mathcal{D}$ is the network diameter and α is greater than the length of the longest chordless cycle in the network.

P_ICorrect can be easily inferred for the asynchronous unison problem since the specification by itself is local: a process p satisfies **P_ICorrect**(p) if and only if the difference between its clock and that of any of its neighbor is at most one increment. Then, *reset*(p) simply consists for p in resetting its clock to 0, and so **P_reset**(p) holds if and only if the clock value of p is 0.

(f,g)-alliance. Recall that (f,g)-alliances are defined in Section 1.5.2.2 (page 21). Ideally, we would like to find a *minimum* (f,g)-alliance. However, this problem is $\mathcal{NP}$-hard, since the minimum $(1,0)$-alliance (*i.e.*, the minimum dominating set problem) is known to be $\mathcal{NP}$-hard [GJ79]. In a best-effort spirit, we can instead consider the problem of finding a 1-*minimal* (f,g)-*alliance*. A is a 1-*minimal* (f,g)-*alliance* if deletion of just one member of A causes A to be no more an (f,g)-*alliance*, *i.e.*, A is an (f,g)-*alliance* but $\forall p \in A, A \setminus \{p\}$ is not an (f,g)-*alliance*.

As explained before, our 1-minimal (f,g)-alliance algorithm is also silent. It works in any arbitrary identified network where the degree of each process p is at least $\max(f(p),g(p))$ (this hypothesis only aims at ensuring the existence of a solution). Its stabilization time is $O(n)$ rounds and $O(\Delta.n.m)$ moves, where m is the number of edges in the network. To the best of our knowledge, until now there was no self-stabilizing algorithm solving the problem without assuming conditions on the relation between f and g. Actually, we have previously proposed an efficient (safely converging) silent self-stabilizing solution [CDD+15] to the minimal (f,g)-alliance problem in an arbitrary identified network for the case where every node p has

a degree at least $g(p)$ and satisfies $f(p) \geq g(p)$. Now, under these settings, the minimal and 1-minimal (f,g)-alliance problems are equivalent [DPRS11].

Again, the requirements of Section 5.2.2.2, necessary to use our reset approach, can be easily obtained since the 1-minimal (f,g)-alliance problem is specified as a conjunction of local predicates at each process (*i.e.*, predicates involving the process and its neighbors).

5.2.2.4 Perspective

An important drawback of our solution is that its memory requirement is not bounded. Actually, in the reset algorithm we use the distance to obtain acyclicity. However, to obtain time-efficiency we do not bound the domain of distance values and do not enforce the algorithm to minimize the values of the distance variables of processes initially involved in a reset.

Hence, the immediate perspective of this work is to try to implement a memory-efficient version of our cooperative reset while keeping its time efficiency.

Chapter 6

Perspectives

In this chapter, I present some new research axes (of course related to self-stabilization) that I feel to be promising and that I plan to investigate in the coming years. These perspectives are presented following three main topics: handling more uncertain environments, more accurately capturing efficiency, and targeting more general solutions.

6.1 Toward More Uncertainty

Several possible sources of uncertainty have still been poorly investigated in the self-stabilizing contexts. Few of them are discussed below.

6.1.1 Unidirectional Networks

The vast majority of the self-stabilizing literature consider bidirectional networks, with our works [BDGPBT09, BDGPB$^+$10] as a notable exception. Now, unidirectional networks, where the possibility of sending information to some process does not necessarily imply the possibility of receiving information from that process, make sense in practical contexts such as wireless sensor networks. Indeed, due to the heterogeneity of the radio supplies, antennas may have different ranges making the network topology unidirectional. The maybe asymmetric communication between neighboring nodes often makes problems more complicated. As a illustrative example, vertex-coloring in the atomic-state model under a central daemon can be solved using local memories depending on local parameters only (namely, the degree of the node; see [ADDP19a]) in bidirectional networks, while the same problem requires at least as many states as the number of processes in the unidirectional cases [BDGPBT09]. For many other tasks such as the asynchronous unison problem for example, it seems that assumptions on the topology (*e.g.*, strong connectivity) are necessary. Furthermore, some problems, such as spanning tree constructions, are not always defined in arbitrary connected unidirectional topologies. Hence, a preliminary important task is to properly define the minimum assumptions for which the considered problem is self-stabilizingly solvable in unidirectional topologies.

6.1.2 Homonymous Systems

Many problems have no deterministic self-stabilizing solution in anonymous networks (with some notable exceptions; see, *e.g.*, [DP12, ADDP19b]). To circumvent these impossibilities, a bunch of work [ACD$^+$17b, DDL19, DIJ19] has been done on *rooted* or *identified* networks to break the symmetries and so allow the design of deterministic solutions. These two latter kinds of systems are computationally equivalent in terms

of expressiveness. An opposite approach has consisted in proposing randomized algorithms, hence achieving a variant of self-stabilization such as the probabilistic self-stabilization; see, *e.g.*, [BDGPB$^+$10]. Recently, we have worked on homonymous systems [ADD$^+$16, ADD$^+$17a], an intermediate model between the (fully) anonymous and (fully) identified ones. In this model, each process has an identifier which may not be unique. Let $\mathcal{L}$ be the set of identifiers present in a system of n processes. Then, the case $|\mathcal{L}| = 1$ (resp., $|\mathcal{L}| = n$) corresponds to the fully anonymous (resp., fully identified) model. Even if the homonymous assumption has some practical justifications, *e.g.*, *group* or *ring signatures* [CvH91, RST01], the main interest of this model is to try to tightly define the limit between the impossibility and the possibility of deterministically solve a task. Now, until now homonymous systems have never been investigated in a self-stabilizing context. In a problem such as leader election, the difficulty of self-stabilization comes from the ability to remove fake identifiers (*i.e.*, values wrongly considered by some processes as process identifiers). This issue seems to be even more difficult to handle when several processes have the same identifier.

6.1.3 High Dynamics

Many of today's networks are highly dynamic, *e.g.*, MANET (*Mobile Ad-Hoc Networks*), VANET (*Vehicular Ad-Hoc Networks*), and DTN (*Delay-Tolerant Networks*), to only quote a few. In such systems, the frequency of topological changes is too high to consider such events as transient events. In other words, when the network dynamics is too high, it should be no more considered as an anomaly but rather as an integral part of the system nature. Hence, self-stabilization in such systems should allow the effective convergence despite the maybe numerous topological changes occurring during the stabilization phase. Ensuring such a convergence is very challenging but the benefits of self-stabilizing solutions in a highly dynamic context seem to be obvious since permanently maintaining a strong safety is simply unrealistic in many cases. Actually, we have already started investigating self-stabilization in an highly dynamic context by modeling it in terms of *Time-Varying Graphs (TVGs)*; see our preliminary results in [ADD$^+$19]. By analogy with partial synchronous systems (see, *e.g.*, [DDF10]), we have defined various classes of TVGs, and we try to capture the weakest temporal connectivity assumptions allowing to solve pseudo- and self- stabilizing leader elections in the synchronous message passing model.

TVGs model systems where dynamicity relates to unpredictable frequent topological changes. Another kind of dynamicity deals with mobile computing entities (agents or swarms of robots), where dynamicity of the system is due to the movements of the computing entities themselves. In this context, research focuses on weakening the resource and computational power of those robots. Typically, in mobile computing, robots have few states, limited computational and communication power, *i.e.*, they are opaque, deaf-mute, myopic, oblivious or endowed with only a few lights, ... In such systems, classical problems are self-organizing tasks, *e.g.*, gathering, flocking, and polygon formation. For these problems, safety conditions are weak and even sometimes vacuum (*e.g.* except the absence of deadlock, there is no safety condition in the perpetual exploration). Hence, self-stabilizing techniques that essentially focus on liveness and convergence are very relevant for solving such problems. In particular, there is today a growing interest on luminous robots [Pel05]. Importing self-stabilizing techniques in such systems seems to be a promising approach. Indeed, many self-stabilizing solutions require a few number of states [Her92a, Joh97, BDPV99a]. Moreover, self-stabilization is often considered as a lightweight fault tolerance technique [Tix06], and so can be achieved in systems that are very weak in terms of computational and communication power.

Finally, the two approaches can be combined, like in the work of Bournat *et al.* [BDP17], by considering robots moving in a TVG. In this paper, authors investigate the exploration of a TVG consisting in a dynamic ring by a team of synchronous robots. Many works can be done in this new research axis, in particular by considering more general classes of dynamic systems.

6.2 Toward More Efficiency

Below, I propose several research axes I plan to explore to more tightly qualify the efficiency (in a broad sense) of self-stabilizing algorithms.

6.2.1 Full Polynomiality

In the atomic-state model, we have proposed many stabilizing algorithms for various problems (spanning structures, wave algorithms, ...) that work under the distributed unfair daemon. For most of them, we have exhibited polynomial upper bounds on their step and move complexities; see, *e.g.*, [DIJ17, DIJ19, DJ19]. The principles we used to obtain these bounds make round complexities inherently linear in n (the number of processes). Now, the classical nontrivial lower bounds for most of those problems is $\Omega(\mathcal{D})$ rounds, where $\mathcal{D}$ is the network diameter [GT02]. Moreover, in many large-scale networks, the diameter is rather logarithmic on n, so finding solutions achieving round complexities linear in $\mathcal{D}$ is more desirable. Only few asynchronous solutions (still in the atomic-state model) that achieve such an upper bound exist, *e.g.*, the atomic-state version of the Dolev's BFS algorithm [Dol93] given in [DJ16]. Moreover, we have shown that several of them actually have a worst-case execution that is exponential in steps; see [DJ16]. So, it seems that efficiency in rounds and steps are often incompatible goals. In a best-effort spirit, Cournier *et al.* [CRV19] have proposed to study what they call *fully polynomial* stabilizing solutions, *i.e.*, stabilizing algorithms whose round complexity is polynomial on the network diameter and step complexity is polynomial on the network size. As an illustrative example, they have proposed a silent self-stabilizing BFS spanning tree algorithm that stabilizes in $O(n^6)$ steps and $O(\mathcal{D}^2)$ rounds in a rooted connected network. The question of generalizing their approach to identified non-rooted networks remains open. As a preliminary step, it is worth investigating whether there exists a fully polynomial self-stabilizing solution for the leader election in arbitrary connected identified networks.

6.2.2 Competitiveness

Silent self-stabilizing algorithms are dedicated to distributed structures such as spanning tree or clustering. Many clustering algorithm are based on dominating sets or generalizations of them. Usually for these latter, the quality of the computed solution is enforced by focusing on inclusion-wise maximality or minimality [DMT09, DDL19]. Inclusion-wise maximality or minimality also exists in many other problems, including for example alliance-related problems [CDD$^+$15]. However, inclusion-wise properties often lead to local minima that can be far from being optimal. For example, in a star network, we can compute a minimal dominating set of size $n-1$ while the optimum is one; see Figure 6.1. On the other hand, computing optimal solutions is often intractable [GJ79].

An in-between approach consists in proposing competitive algorithms, *i.e.*, algorithms which offer provable guarantees on the distance of the computed solution to the optimal one. We have proposed time and space efficient competitive solutions for the maximum leaf spanning tree construction [DKKT10] and the k-clustering problem [DDH$^+$16]. Similarly to the k-clustering problem, an interesting line of research consists in studying competitiveness in the context of parametric problems such as (f,g)-alliances in order to obtain more versatile efficient solutions.

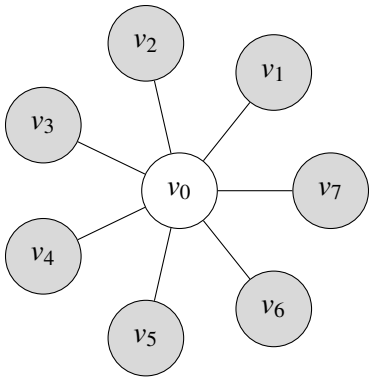


Figure 6.1: Example of degenerated minimal dominating set: the set of grey nodes.

6.2.3 Variants of Self-stabilization

As explained all along this report, defining variants of self-stabilization and proposing solutions achieving them allow to sharply qualify the fault-tolerance property of an algorithm and to explore the limits of expressiveness.

An interesting approach to propose such fine-grained properties has been to propose variants of self-stabilization that are actually a mix of a weak and a strong variant of self-stabilization, *e.g.*, ftps [DDF10] and probabilistic snap-stabilization [AD17]. Indeed, this allows to circumvent impossibility results while keeping interesting fault-tolerant guarantees. Many such variants can be envisioned such as, for example, k-snap-stabilization that would be a self-stabilizing algorithm which is additionally snap-stabilizing when the system is hit by at most k faults.

Then, some existing variants should be further studied. For example, few works have dealt with snap-stabilization in message passing systems [DDNT10, LMV16]. The result of Levé *et al.* [LMV16], in combination with our transformer [CDD$^+$16], implies that snap-stabilization is power equivalent to self-stabilization in message passing systems when the links are both FIFO and of known bounded capacity. However, efficient solutions to dedicated important problems, such as token circulation, still need to be investigated. Moreover, snap-stabilization should be explored in even more general system, *e.g.*, by removing the FIFO assumption or considering topological changes.

Another interesting approach is speculative self-stabilization [DG13]. Speculation aims at guaranteeing that the system satisfies its requirements for all executions (in our context, the fact that the system is self-stabilizing), but also exhibits significantly better performances (here in terms of stabilization time) in a subset of more probable executions. In a recent work [ADD$^+$19], we have initiated research on self-stabilization in highly dynamic identified message-passing systems where the dynamics is modeled using Time-Varying Graphs (TVGs) to obtain solutions tolerating both transient faults and high dynamics. Our goal has been to propose self-stabilizing leader election algorithms in very general classes of TVGs. Now, in very general TVG classes (*e.g.*, in the class of TVGs with recurrent temporal connectivity only), the stabilization time cannot be bounded. In this context, speculation becomes attractive: when the stabilization time cannot be bounded in a very general class, we should try to exhibit an important subclass (*e.g.*, in which we have some bounds on temporal reachability) where stabilization time can be bounded. In such systems, the tight limit where we go from the possibility to the impossibility of bounding the stabilization time is still unknown.

6.2.4 Studying the Average Case

By definition, the stabilization time is impacted by worst case scenarios which are unlikely in practice. So, in many cases, the average-case time complexity may be a more accurate measure of performance assuming a probabilistic model. However, the arbitrary initialization, the asynchronism, the maybe arbitrary network topology, and the algorithm design itself often make the probabilistic analysis intractable. In contrast, another popular approach consists in empirically evaluating the average-case time complexity via simulations. A simulation tool is also of prime interest since it allows testing to find flaws early in the design process.

The empirical study of average time complexities, notably the stabilization time, is most of the time missing in the complexity analysis of self-stabilizing algorithm. Homemade simulations are often restrictive and the confidence on the obtained results is limited. Hence, a simulator dedicated to self-stabilizing algorithm is necessary. We have proposed such a tool: *SASA* [ADJ20][1] (for SimulAtor of Self-stabilizing Algorithms) is an open-source, versatile, lightweight (in terms of memory footprint), and efficient (in terms of simulation time) simulator dedicated to distributed self-stabilizing algorithms written in the atomic-state model. SASA allows batch simulations to perform simulation campaigns. It also includes an interactive graphical simulation environment and a debugger to help the user during the early stages of algorithmic design.

We are currently augmenting SASA with tools to evaluate self-stabilizing solution in terms of fault-containment. We also plan to develop heuristics based on potential functions to exhibit worst-case executions, which are difficult to automatically construct using (uniform) randomization. Again, exhibiting worst-case possible executions is often missing in the analysis of self-stabilizing algorithms. Moreover, when a worst-case analysis has been led, the question about whether the worst-case is likely is often simply skipped. Actually, the time complexity analysis usually rather focuses on upper bounds, and the question of whether or not the proposed bound is tight is often left open. In the same line, amortized cost could be useful, *e.g.*, for resource allocation problems, but is very unlikely in the self-stabilizing literature (with the work of Cournier *et al.* [CDV09b] as a notable exception).

6.3 Toward More Versatility

Below, I present several avenues to develop efficient general schemes for self-stabilization.

6.3.1 Composition Techniques

Compositions are maybe the most popular general techniques used to help the design and proof of complex self-stabilizing algorithms. Several composition methods have been proposed in the literature, *e.g.*, *collateral composition* [GH91, Her92b], *fair composition* [Dol00], *cross-over composition* [BGJ01], and *conditional composition* [DGPV01], to only cite a few. Moreover, we have proposed a variant of the collateral composition called the *hierarchical collateral composition* [DLD$^+$13]. These methods are usually provided with sufficient conditions to easily prove the self-stabilization of the composite algorithm and its complexity. For example, using the hierarchical collateral composition, one can easily derives the stabilization time in rounds of the composite algorithm. The use of this composition technique has been very fruitful since we could establish the self-stabilization (under the distributed weakly fair daemon) and a stabilization time in rounds linear on the size of the network for many complex algorithms written in the atomic-state model; see [DLD$^+$13, DLDR13, DDH$^+$16]. This is probably due to the simplicity of the condition we have proposed. However, this simplicity implies some drawbacks: the stabilization is established under a weakly fair daemon only and

[1] `http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/sasa`

at least one of the two sub-algorithms must be silent. Moreover, as a side effect, time complexity can be shown in rounds only.

Hence, an immediate and important perspective is to propose a composition operator close to the hierarchical collateral composition that offers tight complexity bounds in rounds, steps, and moves; and that is general enough to be able to handle silent tasks as well as mono- and multi- initiator waves. Another important issue is to try to keep the sufficient condition as simple as possible. Notice that, using the hierarchical collateral composition, stabilization time in rounds is somehow additive: it is upper bounded by a sum of the stabilization time of the first algorithm and the stabilization time of the second one starting from a legitimate configuration of the first one. For the move and step complexities, we can expect the stabilization time to be multiplicative, allowing them to prove polynomial bounds in usual cases.

6.3.2 Model Conversions

6.3.2.1 From the Atomic-State to the Register model

Strong models are motivated by (1) the easier way to both design and prove the correctness of algorithms and (2) the existence of model conversions, *i.e.*, general techniques to simulate the behavior of an algorithm written in the strong model into a weaker one and so obtaining a solution achieving the same specification yet in a more general context.

The self-stabilizing conversion from the register model to the message passing model is straightforward using a self-stabilizing data-link algorithm (with the appropriate assumptions). Moreover, notice that this transformation is efficient in both time and space, since, for example, each operation (read or write) in the register model can be emulated in constant time and with a space overhead polynomial on the link capacity; see [ADDP19a] for further details.

Then, Dolev [Dol00] has proposed a self-stabilizing conversion from the atomic-state to the register model. The overall idea is to emulate the atomic-state model in the register model using a self-stabilizing token-based mutual exclusion algorithm in which the critical section (which is performed atomically) consists for the token holder of reading the states of its neighbors and then executing an enabled action, if any. Even if no complexity analysis of the method is given, we can claim that the performances of the output algorithm are weak since it is fully sequential.

Hence, a natural extension consists in proposing a more efficient approach where conflicts between enabled processes will be managed locally rather than globally using a sequential mechanism.

6.3.2.2 From Distance 1 to Distance k

Even in the high-level atomic-state model, proofs may be complex and subtle. Hence, some higher level models have been studied. For example, Gairing *et al.* [GGH$^+$04] have studied a stronger version of atomic-state model, where (1) processes can read at distance two and (2) no two processes can change their state simultaneously. In particular, they designed a method preserving self-stabilization to translate an algorithm written in that distance-2 model into the (fully) asynchronous distance-1 model. For local problems (such as dominating sets), their method allows to easily design and prove self-stabilizing algorithms with a round complexity linear in n and a move complexity polynomial in n.

To further generalize this approach we can study a distance-k model, with $k \geq 2$. More precisely, the expected outcome would be a method to efficiently translate self-stabilizing algorithms written in the sequential distance-k model to algorithms that are self-stabilizing (for the same specification) into the (fully) asynchronous distance-1 model. A natural approach to implement this method is to use an asynchronous

unison algorithm [DP12]. A preliminary promising step has been to propose a k-clustering algorithm where an asynchronous unison is used to collect information at distance k [DDL19].

6.3.3 A General Definition of Resource Allocation Problems

The formalism used to define the LRA problem is simple (the binary relation) and powerful since it allows to express many local resource allocation problems such as dining philosophers, local readers-writers, local mutual exclusion, and local group mutual exclusion. However, not all local resource allocation problems, *e.g.*, local ℓ-exclusion and local k-out-of-ℓ-exclusion, can be described using this formalism. Hence, it is worth finding a new formalism that would allow to represent all local resource allocation problems, while of course remaining simple. Again simplicity is an important objective since it is very helpful for the design of a general algorithm.

Another interesting approach would be to create a simple formalism unifying both local and global resource allocation problems. Then, we could expect to design a general solution for resource allocation problems. Yet, to be interesting, such a generalization should not lead to an inefficient solution in terms of concurrency, availability, or stabilization time.

6.3.4 Meta Theorems and Certification

With a bit of experience, one can identify typical tricks used in the design of self-stabilizing algorithms. In the spirit of our work on the acyclic strategy [ADD18], we can try to give a (quasi-)syntactic definition of algorithms to directly derive correctness proofs. On the other hand, one can try to automatically derive a self-stabilizing algorithm starting from its specification and using the automatic synthesis [FBTK18].

Similarly, typical proof schemes are used in self-stabilizing proofs, *e.g.*, induction [ADD18], potential functions for algorithms [ACD17a], indistinguishability and symmetries for impossibility proofs [DDF10, ADD$^+$19]. Hence, the idea of proving meta-theorems about convergence, partial correctness, ... Such general results can also be certified using a proof assistant (such as Coq [The12]) that checks and validates them mechanically. Such a certification increases the confidence on the truth of the result. Certification of theorems becomes necessary in distributed computing since, due to scientific progress, researchers work on more and more complex and adversarial environments and consider more and more intricate algorithms, and such a complexity may lead to errors, often due to a lack of formalization. We have initiated a framework on certification of self-stabilizing algorithms [ACD17a]. In particular, using this framework, we have certified the sufficient condition related to our hierarchical collateral composition operator [ACD19].

Overall, I think that the use of methods coming from computer-aided verification (model-checking, certification, test, synthesis) will increasingly grow in the distributed computing area in the future since they are very helpful at various levels of the algorithmic design (debugging, proof, and evaluation).

6.3.5 Proof Labeling Schemes

Proof-labeling schemes naturally find applications in self-stabilization. For example, in [BFP14], authors use this concept to design silent self-stabilizing algorithms with bounded memory per process. Based on this approach, they show that every finite task has a silent self-stabilizing algorithm which converges within a linear number of rounds in an arbitrary identified network. However, the step (and move) complexity analysis is simply given up.

A proof-labeling scheme has been also successfully used to implement space-efficient self-stabilizing solution, *i.e.*, Kutten and Trehan have proposed a space-efficient ($O(\log n)$ bits per process) silent DFS

spanning tree construction that stabilizes in $O(n)$ rounds [KT14].

Many general methods presented here actually use local predicates to locally detect inconsistency and so use a kind of underlying proof labeling scheme. The work of Blin *et al.* have shown the power of this method in the context of finite task. Yet, proof labeling schemes in the context of dynamic specification, such as waves for example, still need to be investigated. Finally, it would be interesting to find the generalization based on the proof labeling schemes that would allow to manage both dynamic and static specification.

6.3.6 Rooted versus Identified Networks

As explained before, there exist only a few self-stabilizing solutions that achieve a stabilization time linear on the network diameter $\mathcal{D}$, *i.e.*, [Dol93, DJ16]. Such solutions usually assume the existence of a leader (also called root) in the network.

Connected rooted networks and connected identified ones are computationally equivalent when considering the self-stabilization in the atomic-state model under the distributed unfair daemon. Indeed, a leader can be elected in a connected identified network using, for example, our solution [ACD$^+$17b]. On the other hand, in connected rooted networks, unique process identifiers can be computed by first building a spanning tree using, for example, the atomic-state version of the Dolev's algorithm we have studied in [DJ16]. Then, the processes can be identified using, for example, their ranks in the preorder traversal of the tree.

However, still considering atomic-state model with the distributed unfair daemon, the question of the equivalence of these two assumptions in terms of efficiency remains open. Indeed, in these settings, a BFS spanning tree can be computed in $O(\mathcal{D})$ rounds and a bounded memory per process [DJ16] in any connected rooted network. Then, the preorder ranking in the tree can be computed within $O(\mathcal{D})$ additional rounds still using bounded local memories, thanks to top-down and bottom-up actions along the tree; see [DLDR13]. Hence, overall a connected rooted network can be identified in $O(\mathcal{D})$ rounds and bounded local memories in the atomic-state model under the distributed unfair daemon. However, until now there are no positive or negative results about self-stabilizingly electing a leader in $O(\mathcal{D})$ rounds in a connected identified network under these settings.

However, notice that Kravchik and Kutten [KK13] have partially answered that open question. Indeed, they have proposed in the atomic-state model a self-stabilizing algorithm that elects a leader and computes a spanning tree rooted at the leader node in an identified connected network in $O(\mathcal{D})$ rounds and using bounded process memories, yet assuming a *synchronous* daemon. Their solution is actually based on an asynchronous unison algorithm that stabilizes in $O(\mathcal{D})$ rounds in synchronous anonymous networks assuming the processes know an upper bound on n, the number of processes (*n.b.*, the time complexity of the algorithm does not depend on that upper bound). Leader election and asynchronous unison seem then to be intricately related since the question of implementing in the atomic-state model under the distributed unfair daemon assumption a self-stabilizing unison that stabilized in $O(\mathcal{D})$ rounds using bounded process memories in an anonymous network is still open. Now, such a solution would allow to solve the self-stabilizing asynchronous leader election in identified connected networks in $O(\mathcal{D})$ rounds, using the same approach as Kravchik and Kutten [KK13].

Index

Bibliography

[AB93] Yehuda Afek and Geoffrey M. Brown. Self-Stabilization Over Unreliable Communication Media. *Distributed Computing*, 7(1):27–34, 1993.

[ACD17a] Karine Altisen, Pierre Corbineau, and Stéphane Devismes. A Framework for Certified Self-Stabilization. *Logical Methods in Computer Science*, 13(4), 2017.

[ACD+17b] Karine Altisen, Alain Cournier, Stéphane Devismes, Anaïs Durand, and Franck Petit. Self-Stabilizing Leader Election in Polynomial Steps. *Information and Computation*, 254, Part 3:330 – 366, 2017.

[ACD19] Karine Altisen, Pierre Corbineau, and Stéphane Devismes. Squeezing Streams and Composition of Self-Stabilizing Algorithms. In *FORTE 2019 - 39th International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, Lyngby, Danemark, June 17-21 2019. to appear.

[AD17] Karine Altisen and Stéphane Devismes. On probabilistic snap-stabilization. *Theoretical Computer Science*, 688:49 – 76, 2017. Distributed Computing and Networking.

[ADD+16] Karine Altisen, Ajoy Kumar Datta, Stéphane Devismes, Anaïs Durand, and Lawrence L. Larmore. Leader Election in Rings with Bounded Multiplicity (Short Paper). In Borzoo Bonakdarpour and Franck Petit, editors, *Stabilization, Safety, and Security of Distributed Systems - 18th International Symposium, SSS 2016, Lyon, France, November 7-10, 2016, Proceedings*, volume 10083 of *Lecture Notes in Computer Science*, pages 1–6, 2016.

[ADD+17a] Karine Altisen, Ajoy Kumar Datta, Stéphane Devismes, Anaïs Durand, and Lawrence L. Larmore. Leader Election in Asymmetric Labeled Unidirectional Rings. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*, pages 182–191. IEEE Computer Society, 2017.

[ADD17b] Karine Altisen, Stéphane Devismes, and Anaïs Durand. Concurrency in snap-stabilizing local resource allocation. *J. Parallel Distrib. Comput.*, 102:42–56, 2017.

[ADD18] Karine Altisen, Stéphane Devismes, and Anaïs Durand. Acyclic Strategy for Silent Self-stabilization in Spanning Forests. In Taisuke Izumi and Petr Kuznetsov, editors, *Stabilization, Safety, and Security of Distributed Systems - 20th International Symposium, SSS 2018, Tokyo, Japan, November 4-7, 2018, Proceedings*, volume 11201 of *Lecture Notes in Computer Science*, pages 186–202. Springer, 2018.

[ADD+19] Karine Altisen, Stéphane Devismes, Anaïs Durand, Colette Johnen, and Franck Petit. Self-stabilizing Systems in Spite of High Dynamics. Research report, LaBRI, CNRS UMR 5800, November 2019.

[ADDP19a] Karine Altisen, Stéphane Devismes, Swan Dubois, and Franck Petit. *Introduction to Distributed Self-Stabilizing Algorithms*, volume 8 of *Synthesis Lectures on Distributed Computing Theory*. Morgan & Claypool Publishers (edited by Michel Raynal), 2019.

[ADDP19b] Karine Altisen, Stéphane Devismes, Anaïs Durand, and Franck Petit. Gradual stabilization. *J. Parallel Distrib. Comput.*, 123:26–45, 2019.

[ADGFT08] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. On implementing Omega in systems with weak reliability and synchrony assumptions. *Distributed Computing*, 21(4):285–314, 2008.

[ADGL12] Karine Altisen, Stéphane Devismes, Antoine Gerbaud, and Pascal Lafourcade. Analysis of Random Walks using Tabu Lists. In Magnus M. Halldorsson and Guy Even, editors, *19th International Colloquium on Structural Information and Communication Complexity (SIROCCO'2012)*, LNCS, pages 254–266, Reykjavík, Iceland, June 30 - July 2 2012. Springer.

[ADGL14] Karine Altisen, Stéphane Devismes, Antoine Gerbaud, and Pascal Lafourcade. Comparison of Mean Hitting Times for a Degree-Biased Random Walk. *Discrete Applied Mathematics*, 170:104–109, 2014.

[ADJ20] Karine Altisen, Stéphane Devismes, and Erwan Jahier. SASA: a SimulAtor of Self-stabilizing Algorithms. In Wolfgang Ahrendt and Heike Wehrheim, editors, *Tests and Proofs - 14th International Conference, TAP@STAF 2020, Bergen, Norway, June 22-23, 2020, Proceedings [postponed]*, volume 12165 of *Lecture Notes in Computer Science*, pages 143–154. Springer, 2020.

[ADJL17] Karine Altisen, Stéphane Devismes, Raphaël Jamet, and Pascal Lafourcade. SR3: secure resilient reputation-based routing. *Wireless Networks*, 23(7):2111–2133, Oct 2017.

[ADLP11] Karine Altisen, Stéphane Devismes, Pascal Lafourcade, and Clément Ponsonnet. Routage par marche aléatoire à listes tabous. In *13es Rencontres Francophones sur les Aspects Algorithmiques de Télécommunications (AlgoTel)*, page 14, Cap Estérel, France, May 2011.

[AG94] Anish Arora and Mohamed G. Gouda. Distributed Reset. *IEEE Trans. Computers*, 43(9):1026–1038, 1994.

[AH93] Efthymios Anagnostou and Vassos Hadzilacos. Tolerating Transient and Permanent Failures (Extended Abstract). In *WDAG*, pages 174–188, 1993.

[AKM+93] Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. Time Optimal Self-stabilizing Synchronization. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 652–661. ACM, 1993.

[AN05] Anish Arora and Mikhail Nesterenko. Unifying stabilization and termination in message-passing systems. *Distributed Computing*, 17(3):279–290, 2005.

[AO94] Baruch Awerbuch and Rafail Ostrovsky. Memory-Efficient and Self-Stabilizing Network RESET (Extended Abstract). In James H. Anderson, David Peleg, and Elizabeth Borowsky, editors, *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, Los Angeles, California, USA, August 14-17, 1994*, pages 254–263. ACM, 1994.

[APSV91] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by Local Checking and Correction (Extended Abstract). In *Proceedings of the 32Nd Annual Symposium on Foundations of Computer Science*, SFCS '91, pages 268–277. IEEE Computer Society, 1991.

[APVD94] Baruch Awerbuch, Boaz Patt-Shamir, George Varghese, and Shlomi Dolev. Self-Stabilization by Local Checking and Global Reset (extended abstract). In *Distributed Algorithms, 8th International Workshop, WDAG*, pages 326–339, 1994.

[AS85] Bowen Alpern and Fred B. Schneider. Defining Liveness. *Information Processing Letters*, 21(4):181–185, 1985.

[BCT96] Anindya Basu, Bernadette Charron-Bost, and Sam Toueg. Simulating Reliable Links with Unreliable Links in the Presence of Process Crashes. In Özalp Babaoglu and Keith Marzullo, editors, *Distributed Algorithms, 10th International Workshop, WDAG '96*, volume 1151 of *Lecture Notes in Computer Science*, pages 105–122, 1996.

[BCV03] Lélia Blin, Alain Cournier, and Vincent Villain. An Improved Snap-Stabilizing PIF Algorithm. In Shing-Tsaan Huang and Ted Herman, editors, *Self-Stabilizing Systems, 6th International Symposium, SSS 2003, San Francisco, CA, USA, June 24-25, 2003, Proceedings*, volume 2704 of *Lecture Notes in Computer Science*, pages 199–214. Springer, 2003.

[BDGPB+10] Samuel Bernard, Stéphane Devismes, Maria Gradinariu Potop-Butucaru, Katy Paroux, and Sébastien Tixeuil. Probabilistic Self-Stabilizing Vertex Coloring in Unidirectional Anonymous Networks. In *ICDCN'2010, 11th International Conference on Distributed Computing and Networking*, pages 167–177, Kolkata, India, January 2010.

[BDGPBT09] Samuel Bernard, Stéphane Devismes, Maria Gradinariu Potop-Butucaru, and Sébastien Tixeuil. Optimal deterministic self-stabilizing vertex coloring in unidirectional anonymous networks. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–8, Roma, Italia, 2009. IEEE Computer Society.

[BDP16] Borzoo Bonakdarpour, Stéphane Devismes, and Franck Petit. Snap-Stabilizing Committee Coordination. *Journal of Parallel and Distributed Computing (JPDC)*, 87:26–42, 2016.

[BDP17] Marjorie Bournat, Swan Dubois, and Franck Petit. Computability of Perpetual Exploration in Highly Dynamic Rings. In Kisung Lee and Ling Liu, editors, *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*, pages 794–804. IEEE Computer Society, 2017.

[BDPV99a] Alain Bui, Ajoy K. Datta, Franck Petit, and Vincent Villain. State-optimal snap-stabilizing PIF in tree networks. In Anish Arora, editor, *Workshop on Self-stabilizing Systems*, pages 78–85. IEEE Computer Society, 1999.

[BDPV99b] Alain Bui, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. Snap-Stabilizing PIF Algorithm in Trees. In Cyril Gavoille, Jean-Claude Bermond, and André Raspaud, editors, *SIROCCO'99, 6th International Colloquium on Structural Information & Communication Complexity, Lacanau-Ocean, France, 1-3 July, 1999*, pages 32–46. Carleton Scientific, 1999.

[BDPV07] A. Bui, A. K. Datta, F. Petit, and V. Villain. Snap-Stabilization and PIF in Tree Networks. *Distributed Computing*, 20(1):3–19, 2007.

[BF14] Lélia Blin and Pierre Fraigniaud. Polynomial-Time Space-Optimal Silent Self-Stabilizing Minimum-Degree Spanning Tree Construction. *CoRR*, abs/1402.2496, 2014.

[BFP14] Lélia Blin, Pierre Fraigniaud, and Boaz Patt-Shamir. On Proof-Labeling Schemes versus Silent Self-stabilizing Algorithms. In *16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2014), Springer LNCS 8756*, pages 18–32, 2014.

[BGJ01] Joffroy Beauquier, Maria Gradinariu, and Colette Johnen. Cross-Over Composition - Enforcement of Fairness under Unfair Adversary. In *5th International Workshop on Self-Stabilizing Systems, (WSS 2001), Springer LNCS 2194*, pages 19–34, 2001.

[BGJ07] Joffroy Beauquier, Maria Gradinariu, and Colette Johnen. Randomized self-stabilizing and space optimal leader election under arbitrary scheduler on rings. *Distributed Computing*, 20(1):75–93, 2007.

[BGK98] Joffroy Beauquier, Christophe Genolini, and Shay Kutten. *k*-Stabilization of Reactive Tasks. In Brian A. Coan and Yehuda Afek, editors, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, page 318. ACM, 1998.

[BGM93] James E. Burns, Mohamed G. Gouda, and Raymond E. Miller. Stabilization and Pseudo-Stabilization. *Distributed Computing*, 7(1):35–42, 1993.

[BK97] Joffroy Beauquier and Synnöve Kekkonen-Moneta. On FTSS-Solvable Distributed Problems. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, page 290, 1997.

[BLP08] Christian Boulinier, Mathieu Levert, and Franck Petit. Snap-Stabilizing Waves in Anonymous Networks. In Shrisha Rao, Mainak Chatterjee, Prasad Jayanti, C. Siva Ram Murthy, and Sanjoy Kumar Saha, editors, *Proceedings of Distributed Computing and Networking, 9th International Conference (ICDCN), Kolkata, India, January 5-8, 2008*, volume 4904 of *Lecture Notes in Computer Science*, pages 191–202. Springer, 2008.

[BO83] Michael Ben-Or. Another Advantage of Free Choice (Extended Abstract): Completely Asynchronous Agreement Protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, pages 27–30, New York, NY, USA, 1983. ACM.

[BPV04] Christian Boulinier, Franck Petit, and Vincent Villain. When graph theory helps self-stabilization. In Soma Chaudhuri and Shay Kutten, editors, *Proceedings of the Twenty-Third Annual Symposium on Principles of Distributed Computing (PODC), St. John's, Newfoundland, Canada, July 25-28, 2004*, pages 150–159. ACM, 2004.

[CD94] Zeev Collin and Shlomi Dolev. Self-Stabilizing Depth-First Search. *Inf. Process. Lett.*, 49(6):297–301, 1994.

[CDD+15] Fabienne Carrier, Ajoy Kumar Datta, Stéphane Devismes, Lawrence L. Larmore, and Yvan Rivierre. Self-Stabilizing (f,g)-Alliances with Safe Convergence. *Journal of Parallel and Distributed Computing (JPDC)*, 81-82:11–23, 2015.

[CDD+16] Alain Cournier, Ajoy Kumar Datta, Stéphane Devismes, Franck Petit, and Vincent Villain. The Expressive Power of Snap-Stabilization. *Theoretical Computer Science (TCS)*, 626:40–66, 2016.

[CDDL15] Fabienne Carrier, Ajoy Kumar Datta, Stéphane Devismes, and Lawrence L. Larmore. Self-Stabilizing ℓ-Exclusion Revisited. In Sajal K. Das, Dilip Krishnaswamy, Santonu Karkar, Amos Korman, Mohan Kumar, Marius Portmann, and Srikanth Sastry, editors, *ICDCN'2015, 16th International Conference on Distributed Computing and Networking*, pages 3:1–3:10, Goa, India, January 4-7 2015. ACM.

[CDP03] Sébastien Cantarell, Ajoy Kumar Datta, and Franck Petit. Self-Stabilizing Atomicity Refinement Allowing Neighborhood Concurrency. In *Self-Stabilizing Systems, 6th International Symposium, SSS 2003, San Francisco, CA, USA, June 24-25, 2003, Proceedings*, pages 102–112, 2003.

[CDPR11] Fabienne Carrier, Stéphane Devismes, Franck Petit, and Yvan Rivierre. Asymptotically Optimal Deterministic Rendezvous. *International Journal of Foundations of Computer Science (IJFCS)*, 22:1143–1159, 2011.

[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'02)*, pages 199–206, 2002.

[CDPV06] Alain Cournier, Stéphane Devismes, Franck Petit, and Vincent Villain. Snap-Stabilizing Depth-First Search on Arbitrary Networks. *The Computer Journal*, 49(3):268–280, 2006.

[CDV06a] Alain Cournier, Stéphane Devismes, and Vincent Villain. From Self- to Snap- Stabilization. In Maria Gradinariu Ajoy Kumar Datta, editor, *SSS'06, 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, volume 4280 of *Lecture Notes in Computer Science*, pages 199–213, Dallas, TX, USA, November 2006. Springer.

[CDV06b] Alain Cournier, Stéphane Devismes, and Vincent Villain. Snap-Stabilizing PIF and Useless Computations. In *ICPADS'06, 12th International Conference on Parallel and Distributed Systems*, pages 39–48, Minneapolis, Minnesota, USA, July 2006. IEEE Computer Society.

[CDV09a] Alain Cournier, Stéphane Devismes, and Vincent Villain. Light enabling snap-stabilization of fundamental protocols. *TAAS, ACM Transactions on Autonomous and Adaptive Systems*, 4(1):6:1–6:27, 2009.

[CDV09b] Alain Cournier, Swan Dubois, and Vincent Villain. A snap-stabilizing point-to-point communication protocol in message-switched networks. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, pages 1–11. IEEE, 2009.

[CFG92] Jean-Michel Couvreur, Nissim Francez, and Mohamed G. Gouda. Asynchronous Unison (Extended Abstract). In *ICDCS*, pages 486–493, 1992.

[CH09] J. A. Cobb and C. T. Huang. Stabilization of Maximal-Metric Routing without Knowledge of Network Size. In *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 306–311, December 2009.

[Cha82] EJH Chang. Echo algorithms: depth parallel operations on general graphs. *IEEE Transactions on Software Engineering*, SE-8:391–401, 1982.

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The Weakest Failure Detector for Solving Consensus. *J. ACM*, 43(4):685–722, 1996.

[CL85] KM Chandy and L Lamport. Distributed snapshots: determining Global States od Distributed Systems. *ACM Transactions on Computers Systems*, 3(1):63–75, 1985.

[CM84] K. Mani Chandy and Jayadev Misra. The Drinking Philosopher's Problem. *ACM Trans. Program. Lang. Syst.*, 6(4):632–646, 1984.

[Cou09a] Alain Cournier. A new polynomial silent stabilizing spanning-tree construction algorithm. In *International Colloquium on Structural Information and Communication Complexity*, pages 141–153. Springer, 2009.

[Cou09b] Alain Cournier. *Graphes et algorithmique distribuée stabilisante*. PhD thesis, Université de Picardie Jules Verne, Amiens, France, 2009.

[CRV19] Alain Cournier, Stephane Rovedakis, and Vincent Villain. The first fully polynomial stabilizing algorithm for BFS tree construction. *Inf. Comput.*, 265:26–56, 2019.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM*, 43(2):225–267, March 1996.

[CvH91] David Chaum and Eugène van Heyst. Group Signatures. In *EUROCRYPT*, pages 257–265, 1991.

[DDF10] Carole Delporte-Gallet, Stéphane Devismes, and Hugues Fauconnier. Stabilizing leader election in partial synchronous systems with crash failures. *Journal of Parallel and Distributed Computing*, 70(1):45–58, 2010.

[DDGFL10] Stéphane Devismes, Carole Delporte-Gallet, Hugues Fauconnier, and Mikel Larrea. Algorithms For Extracting Timeliness Graphs. In Boaz Patt-Shamir and Tinaz Ekim, editors, *17th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2010)*, pages 127–141, Sirince, TURKEY, june 7-11 2010. Springer.

[DDH+16] Ajoy Kumar Datta, Stéphane Devismes, Karel Heurtefeux, Lawrence L. Larmore, and Yvan Rivierre. Competitive Self-Stabilizing k-Clustering. *Theoretical Computer Science (TCS)*, 626:110–133, 2016.

[DDHL11] Ajoy Kumar Datta, Stéphane Devismes, Florian Horn, and Lawrence L. Larmore. Self-stabilizing k-out-of-l exclusion on tree networks. *International Journal of Foundations of Computer Science*, 22(3):657–677, 2011.

[DDL17] Ajoy Kumar Datta, Stéphane Devismes, and Lawrence L. Larmore. Self-stabilizing silent disjunction in an anonymous network. *Theor. Comput. Sci.*, 665:51–72, 2017.

[DDL19] Ajoy Kumar Datta, Stéphane Devismes, and Lawrence L. Larmore. A silent self-stabilizing algorithm for the generalized minimal k-dominating set problem. *Theor. Comput. Sci.*, 753:35–63, 2019.

[DDLT13] Ajoy Kumar Datta, Stéphane Devismes, Lawrence L. Larmore, and Sébastien Tixeuil. Fast Leader (Full) Recovery despite Dynamic Faults. In *ICDCN: 14th International Conference on Distributed Computing and Networking*, Lecture Notes in Computer Science, pages 428–433, Tata Institute of Fundamental Research, Mumbai, India, January 3-6 2013. Springer.

[DDLV17] Ajoy Kumar Datta, Stéphane Devismes, Lawrence L. Larmore, and Vincent Villain. Self-Stabilizing Weak Leader Election in Anonymous Trees Using Constant Memory per Edge. *Parallel Processing Letters*, 27(2):1–18, 2017.

[DDNT10] Sylvie Delaët, Stéphane Devismes, Mikhail Nesterenko, and Sébastien Tixeuil. Snap-Stabilization in Message-Passing Systems. *Journal of Parallel and Distributed Computing (JPDC)*, 70(12):1220–1230, 2010.

[DDT06] Sylvie Delaët, Bertrand Ducourthial, and Sébastien Tixeuil. Self-stabilization with r-operators revisited. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 3(10):498–514, 2006.

[Dev05] Stéphane Devismes. A Silent Self-stabilizing Algorithm for Finding Cut-nodes and Bridges. *Parallel Process. Lett.*, 15(1-2):183–198, 2005.

[Dev10] Stéphane Devismes. *Quelques Contributions à la Stabilisation Instantanée*. Editions universitaires européennes, 2010. In French.

[DG13] S. Dubois and R. Guerraoui. Introducing speculation in self-stabilization: an application to mutual exclusion. In *PODC*, pages 290–298, 2013.

[DGDF+08] Carole Delporte-Gallet, Stéphane Devismes, Hugues Fauconnier, Franck Petit, and Sam Toueg. With Finite Memory Consensus Is Easier Than Reliable Broadcast. In *OPODIS'09, 12th International Conference On Principles of DIstributed Systems*, volume 5401 of *Lecture Notes in Computer Science*, pages 41–57, Luxor, Egypt, december 2008. Springer.

[DGDF10] Carole Delporte-Gallet, Stéphane Devismes, and Hugues Fauconnier. Approximation of δ-timeliness. In Shlomi Dolev, editor, *12th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2010)*, pages 435–451, New York City, USA, September 2010. LNCS.

[DGPV01] Ajoy K. Datta, Shivashankar Gurumurthy, Franck Petit, and Vincent Villain. Self-Stabilizing Network Orientation Algorithms in Arbitrary Rooted Networks. *Studia Informatica Universalis*, 1(1):1–22, 2001.

[DGS99] Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory Requirements for Silent Stabilization. *Acta Informatica*, 36(6):447–462, 1999.

[DH95] Shlomi Dolev and Ted Herman. Superstabilizing Protocols for Dynamic Distributed Systems. *Chicago Journal of Theoretical Computer Science*, 1995.

[DHV03a] Ajoy Kumar Datta, Rachid Hadid, and Vincent Villain. A New Self-Stabilizing k-out-of-l Exclusion Algorithm on Rings. In *Self-Stabilizing Systems, 6th International Symposium, SSS 2003, San Francisco, CA, USA, June 24-25, 2003, Proceedings*, pages 113–128, 2003.

[DHV03b] Ajoy Kumar Datta, Rachid Hadid, and Vincent Villain. A Self-Stabilizing Token-Based k-out-of-l-Exclusion Algorithm. *Concurrency and Computation: Practice and Experience*, 15(11-12):1069–1091, 2003.

[Dij73] Edsger W. Dijkstra. Self-stabilization in Spite of Distributed Control. Technical Report EWD 391, University of Texas, 1973. Published in 1982 as "Selected Writings on Computing: A 0–387–90652–5.

[Dij74] Edsger W. Dijkstra. Self-stabilization in Spite of Distributed Control. *Commun. ACM*, 17(11):643–644, 1974.

[Dij78] Edsger W Dijkstra. Two Starvation-Free Solutions of a General Exclusion Problem. Technical Report EWD 625, Plataanstraat 5, 5671, AL Nuenen, The Netherlands, 1978.

[DIJ17] Stéphane Devismes, David Ilcinkas, and Colette Johnen. Self-Stabilizing Disconnected Components Detection and Rooted Shortest-Path Tree Maintenance in Polynomial Steps. *Discrete Mathematics & Theoretical Computer Science*, 19(3), 2017.

[DIJ19] Stéphane Devismes, David Ilcinkas, and Colette Johnen. Silent self-stabilizing scheme for spanning-tree-like constructions. In R. C. Hansdah, Dilip Krishnaswamy, and Nitin Vaidya, editors, *Proceedings of the 20th International Conference on Distributed Computing and Networking, ICDCN 2019, Bangalore, India, January 04-07, 2019*, pages 158–167, 2019.

[DIJ20] Stéphane Devismes, David Ilcinkas, and Colette Johnen. Optimized Silent Self-Stabilizing Scheme for Tree-based Constructions. Research report, VERIMAG/LaBRI, August 2020. Under submission to Algorithmica.

[DIM93] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity. *Distributed Comput.*, 7(1):3–16, 1993.

[DIM97a] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Resource Bounds for Self-Stabilizing Message-Driven Protocols. *SIAM Journal on Computing*, 26(1):273–290, 1997.

[DIM97b] 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.

[DJ16] Stéphane Devismes and Colette Johnen. Silent self-stabilizing BFS tree algorithms revisited. *J. Parallel Distributed Comput.*, 97:11–23, 2016.

[DJ19] Stéphane Devismes and Colette Johnen. Self-Stabilizing Distributed Cooperative Reset. In *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems (ICDCS 2019)*, Dalles, USA, July 7 - 10 2019. to appear.

[DKKT10] Stéphane Devismes, Hirotsugu Kakugawa, Sayaka Kamei, and Sébastien Tixeuil. A Self-Stabilizing 3-Approximation for the Maximum Leaf Spanning Tree Problem in Arbitrary Networks. *Journal of Combinatorial Optimization (Special Issue)*, 21(1), 2010.

[DLD+13] Ajoy Kumar Datta, Lawrence L. Larmore, Stéphane Devismes, Karel Heurtefeux, and Yvan Rivierre. Self-Stabilizing Small k-Dominating Sets. *IJNC, International Journal of Networking and Computing*, 3(1):116–136, 2013.

[DLDR13] Ajoy Kumar Datta, Lawrence L. Larmore, Stéphane Devismes, and Yvan Rivierre. Self-Stabilizing Labeling and Ranking in Ordered Trees. *Theoretical Computer Science (Special Issue SSS 2011)*, 512:49–66, 2013.

[DLM14] Ajoy Kumar Datta, Lawrence L. Larmore, and Toshimitsu Masuzawa. A Communication-Efficient Self-stabilizing Algorithm for Breadth-First Search Trees. In Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro, editors, *Principles of Distributed Systems - 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings*, volume 8878 of *Lecture Notes in Computer Science*, pages 293–306. Springer, 2014.

[DLP+12] Stéphane Devismes, Anissa Lamani, Franck Petit, Pascal Raymond, and Sébastien Tixeuil. Optimal Grid Exploration by Asynchronous Oblivious Robots. In Christian Scheideler Sukumar Ghosh, editor, *14th International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS*, pages 64–76, Toronto, Canada, October 2012. LNCS.

[DLPT19] Stéphane Devismes, Anissa Lamani, Franck Petit, and Sébastien Tixeuil. Optimal torus exploration by oblivious robots. *Computing*, 101(9):1241–1264, 2019.

[DLS88] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[DMT09] Stéphane Devismes, Toshimitsu Masuzawa, and Sébastien Tixeuil. Communication Efficiency in Self-Stabilizing Silent Protocols. In *ICDCS'09, International Conference on Distributed Computing Systems*, pages 474–481, Montréal, Canada, June 2009. IEEE Computer Society.

[DMT15] Swan Dubois, Toshimitsu Masuzawa, and Sébastien Tixeuil. Maximum Metric Spanning Tree Made Byzantine Tolerant. *Algorithmica*, 73(1):166–201, 2015.

[Dol93] Shlomi Dolev. Optimal time self stabilization in dynamic systems. In André Schiper, editor, *Distributed Algorithms*, pages 160–173, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[Dol00] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

[DP12] Stéphane Devismes and Franck Petit. On efficiency of unison. In Yann Busnel Lélia Blin, editor, *4th Workshop on Theoretical Aspects of Dynamic Distributed Systems, TADDS*, pages 20–25, Roma, Italy, December 17 2012. ACM.

[DPRS11] Mitre Costa Dourado, Lucia Draque Penso, Dieter Rautenbach, and Jayme Luiz Szwarcfiter. The South Zone: Distributed Algorithms for Alliances. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, (SSS)*, volume 6976 of *Lecture Notes in Computer Science*, pages 178–192, Grenoble, France, October 10-12 2011. Springer.

[DPT13] Stéphane Devismes, Franck Petit, and Sébastien Tixeuil. Optimal Probabilistic Ring Exploration by Semi-Synchronous Oblivious Robots. *Theoretical Computer Science (TCS)*, 498:10–27, 2013.

[DPV11a] Stéphane Devismes, Franck Petit, and Vincent Villain. Autour de l'Auto-stabilisation. Partie I : Techniques généralisant l'approche. *Technique et science informatiques (TSI)*, 30/7:883–894, octobre 2011. numéro spécial ALGORITHMIQUE DISTRIBUÉE (in French).

[DPV11b] Stéphane Devismes, Franck Petit, and Vincent Villain. Autour de l'Auto-Stabilisation. Partie II : Techniques spécialisant l'approche. *Technique et science informatiques (TSI)*, 30/7:895–922, octobre 2011. numéro spécial ALGORITHMIQUE DISTRIBUÉE (in French).

[DT02] Sylvie Delaët and Sébastien Tixeuil. Tolerating Transient and Intermittent Failures. *Journal of Parallel and Distributed Computing*, 62(5):961–981, 2002.

[DT11] Swan Dubois and Sébastien Tixeuil. A Taxonomy of Daemons in Self-stabilization. *CoRR*, abs/1110.0334, 2011.

[DTY15] Stéphane Devismes, Sébastien Tixeuil, and Masafumi Yamashita. Weak vs. Self vs. Probabilistic Stabilization. *International Journal of Foundations of Computer Science*, 26(3):293–320, 2015.

[FBA+17] Ali Fahs, Rodolphe Bertolini, Olivier Alphand, Franck Rousseau, Karine Altisen, and Stéphane Devismes. Collision Prevention in Distributed 6TiSCH Networks. In *IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, Rome, Italy, Oct 2017.

[FBTK18] Fathiyeh Faghih, B. Bonakdarpour, Sebastien Tixeuil, and S. Kulkarni. Automated synthesis of distributed self-stabilizing protocols. *Logical Methods in Computer Science*, 14, 01 2018.

[FLBB79] Michael J. Fischer, Nancy A. Lynch, James E. Burns, and Allan Borodin. Resource Allocation with Immunity to Limited Process Failure (Preliminary Report). In *20th Annual Symposium on Foundations of Computer Science*, pages 234–254, San Juan, Puerto Rico, 29-31 October 1979. IEEE Computer Society.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, 1985.

[FYHY14] Lin Fei, Sun Yong, Ding Hong, and Ren Yizhi. Self Stabilizing Distributed Transactional Memory Model and Algorithms. *Journal of Computer Research and Development*, 51(9):2046, 2014.

[GGH+04] Martin Gairing, Wayne Goddard, Stephen T. Hedetniemi, Petter Kristiansen, and Alice A. McRae. Distance-two information in self-stabilizing algorithms. *Parallel Process. Lett.*, 14(3-4):387–398, 2004.

[GGHP07] Sukumar Ghosh, Arobinda Gupta, Ted Herman, and Sriram V. Pemmaraju. Fault-containing self-stabilizing distributed protocols. *Distributed Computing*, 20(1):53–73, 2007.

[GH91] Mohamed G. Gouda and Ted Herman. Adaptive Programming. *IEEE Transactions on Software Engineering*, 17(9):911–921, 1991.

[GH07] Mohamed G. Gouda and F. Furman Haddix. The Alternator. *Distributed Computing*, 20(1):21–28, 2007.

[Gho14] Sukumar Ghosh. *Distributed Systems: An Algorithmic Approach, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2014.

[GJ79] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[GM91] Mohamed G. Gouda and Nicholas J. Multari. Stabilizing Communication Protocols. *IEEE Transactions on Computers*, 40(4):448–458, 1991.

[Gou01] Mohamed G. Gouda. The Theory of Weak Stabilization. In Ajoy K. Datta and Ted Herman, editors, *Self-Stabilizing Systems, 5th International Workshop, WSS 2001*, volume 2194 of *Lecture Notes in Computer Science*, pages 114–123. Springer, 2001.

[GSB94] Rajiv Gupta, Scott A. Smolka, and Shaji Bhaskar. On Randomization in Sequential and Distributed Algorithms. *ACM Comput. Surv.*, 26(1):7–86, 1994.

[GT02] Christophe Genolini and Sébastien Tixeuil. A Lower Bound on Dynamic k-Stabilization in Asynchronous Systems. In *21st Symposium on Reliable Distributed Systems (SRDS)*, pages 211–221, Osaka, Japan, October 13-16 2002. IEEE Computer Society.

[GT07] Maria Gradinariu and Sébastien Tixeuil. Conflict Managers for Self-Stabilization without Fairness Assumption. In *27th IEEE International Conference on Distributed Computing Systems (ICDCS 2007), June 25-29, 2007, Toronto, Ontario, Canada*, page 46, 2007.

[Her90] Ted Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, 1990.

[Her92a] Ted Herman. Self-stabilization: ramdomness to reduce space. *Information Processing Letters*, 6:95–98, 1992.

[Her92b] Ted Richard Herman. *Adaptivity through distributed convergence*. PhD thesis, University of Texas at Austin, 1992.

[Her00] Ted Herman. Superstabilizing Mutual Exclusion. *Distributed Computing*, 13(1):1–17, 2000.

[HM08] Markus C. Huebscher and Julie A. McCann. A Survey of Autonomic Computing: Degrees, Models, and Applications. *ACM Computing Surveys*, 40(3):1–28, 2008.

[HT04] Ted Herman and Sébastien Tixeuil. A Distributed TDMA Slot Assignment Algorithm for Wireless Sensor Networks. In *Algorithmic Aspects of Wireless Sensor Networks: First International Workshop, ALGOSENSORS 2004, Turku, Finland, July 16, 2004. Proceedings*, volume 3121 of *Lecture Notes in Computer Science*, pages 45–58. Springer, 2004.

[Hua00] Shing-Tsaan Huang. The Fuzzy Philosophers. In *Parallel and Distributed Processing, 15 IPDPS 2000 Workshops, Cancun, Mexico, May 1-5, 2000, Proceedings*, pages 130–136, 2000.

[HV01] Rachid Hadid and Vincent Villain. A New Efficient Tool for the Design of Self-Stabilizing l-Exclusion Algorithms: The Controller. In Ajoy Kumar Datta and Ted Herman, editors, *Self-Stabilizing Systems, 5th International Workshop, WSS 2001, Lisbon, Portugal, October 1-2, 2001, Proceedings*, volume 2194 of *Lecture Notes in Computer Science*, pages 136–151. Springer, 2001.

[IJ90] Amos Israeli and Marc Jalfon. Token Management Schemes and Random Walks Yield Self-Stabilizing Mutual Exclusion. In Cynthia Dwork, editor, *Proceedings of the Ninth Annual Symposium on Principles of Distributed Computing (PODC)*, pages 119–131. ACM, 1990.

[JADT02] Colette Johnen, Luc Onana Alima, Ajoy Kumar Datta, and Sébastien Tixeuil. Optimal Snap-Stabilizing Neighborhood Synchronizer in Tree networks. *Parallel Processing Letters*, 12(3-4):327–340, 2002.

[Joh97] Colette Johnen. Memory Efficient, Self-Stabilizing Algorithm to Construct BFS Spanning Trees. In James E. Burns and Hagit Attiya, editors, *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, page 288. ACM, 1997.

[KAD+09] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. L. Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Trans. Comp. Syst.*, 27:1–39, 2009.

[KC99] Mehmet Hakan Karaata and Pranay Chaudhuri. A Self-Stabilizing Algorithm for Bridge Finding. *Distributed Computing*, 12(1):47–53, 1999.

[KC03] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[KK05] Adrian Kosowski and Lukasz Kuszner. A Self-stabilizing Algorithm for Finding a Spanning Tree in a Polynomial Number of Moves. In *6th International Conference Parallel Processing and Applied Mathematics, (PPAM'05), Springer LNCS 3911*, pages 75–82, 2005.

[KK07] Sayaka Kamei and Hirotsugu Kakugawa. A Self-stabilizing Approximation Algorithm for the Minimum Weakly Connected Dominating Set with Safe Convergence. In *Proceedings of the First International Workshop on Reliability, Availability, and Security (WRAS)*, pages 57–67, Paris, France, September 2007.

[KK13] Alex Kravchik and Shay Kutten. Time Optimal Synchronous Self Stabilizing Spanning Tree. In Yehuda Afek, editor, *Distributed Computing*, pages 91–105, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[KKP10] Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Computing*, 22(4):215–233, 2010.

[KM06] Hirotsugu Kakugawa and Toshimitsu Masuzawa. A Self-stabilizing Minimal Dominating Set Algorithm with Safe Convergence. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 263–263, Washington, DC, USA, 2006. IEEE Computer Society.

[KMO11] Fabian Kuhn, Yoram Moses, and Rotem Oshman. Coordinated consensus in dynamic networks. In Cyril Gavoille and Pierre Fraigniaud, editors, *Proceedings of the 30th Annual ACM*

Symposium on Principles of Distributed Computing, PODC 2011, San Jose, CA, USA, June 6-8, 2011, pages 1–10. ACM, 2011.

[KNKM18] Yonghwan Kim, Junya Nakamura, Yoshiaki Katayama, and Toshimitsu Masuzawa. A Cooperative Partial Snapshot Algorithm for Checkpoint-Rollback Recovery of Large-Scale and Dynamic Distributed Systems. In *CANDAR'18*, pages 285–291, 11 2018.

[KP93] Shmuel Katz and Kenneth J. Perry. Self-Stabilizing Extensions for Message-Passing Systems. *Distributed Computing*, 7(1):17–26, 1993.

[KPS99] Shay Kutten and Boaz Patt-Shamir. Stabilizing time-adaptive protocols. *Theoretical Computer Science*, 220(1):93 – 111, 1999.

[KT14] Shay Kutten and Chhaya Trehan. Fast and Compact Distributed Verification and Self-stabilization of a DFS Tree. In *Principles of Distributed Systems - 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings*, pages 323–338, 2014.

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.

[Lam12] Leslie Lamport. How to write a 21st century proof. *Journal of Fixed Point Theory and Applications*, 11(1):43–63, 2012.

[Lan77] Gérard Le Lann. Distributed Systems - Towards a Formal Approach. In *IFIP Congress*, pages 155–160, 1977.

[LFA00] M. Larrea, A. Fernandez, and S. Arevalo. Optimal implementation of the weakest failure detector for solving consensus. In *Proceedings 19th IEEE Symposium on Reliable Distributed Systems SRDS-2000*, pages 52–59, Oct 2000.

[LGW04] Alberto Leon-Garcia and Indra Widjaja. *Communication Networks*. McGraw-Hill, Inc., New York, NY, USA, 2 edition, 2004.

[LMV16] Florence Levé, Khaled Mohamed, and Vincent Villain. Snap-Stabilizing PIF on Arbitrary Connected Networks in Message Passing Model. In Borzoo Bonakdarpour and Franck Petit, editors, *Stabilization, Safety, and Security of Distributed Systems - 18th International Symposium, SSS 2016, Lyon, France, November 7-10, 2016, Proceedings*, volume 10083 of *Lecture Notes in Computer Science*, pages 281–297, 2016.

[Lyn68] William C. Lynch. Computer Systems: Reliable Full-duplex File Transmission over Half-duplex Telephone Line. *Communications of the ACM*, 11(6):407–410, June 1968.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[MA89] Yossi Matias and Yehuda Afek. Simple and Efficient Election Algorithms for Anonymous Networks. In Jean-Claude Bermond and Michel Raynal, editors, *Proceedings of Distributed Algorithms, 3rd International Workshop (WDAG), Nice, France, September 26-28, 1989*, volume 392 of *Lecture Notes in Computer Science*, pages 183–194. Springer, 1989.

[MP90] Zohar Manna and Amir Pnueli. A Hierarchy of Temporal Properties. In Cynthia Dwork, editor, *Proceedings of the Ninth Annual Symposium on Principles of Distributed Computing, Quebec City, Quebec, Canada, August 22-24, 1990*, pages 377–410. ACM, 1990.

[MW87] S. Moran and Y. Wolfstahl. Extended Impossibility Results for Asynchronous Complete Networks. *Inf. Process. Lett.*, 26(3):145–151, November 1987.

[NA02a] Mikhail Nesterenko and Anish Arora. Dining Philosophers that Tolerate Malicious Crashes. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 191–198. IEEE Computer Society, 2002.

[NA02b] Mikhail Nesterenko and Anish Arora. Stabilization-Preserving Atomicity Refinement. *J. Parallel Distrib. Comput.*, 62(5):766–791, 2002.

[NM02] Mikhail Nesterenko and Masaaki Mizuno. A Quorum-Based Self-Stabilizing Distributed Mutual Exclusion Algorithm. *J. Parallel Distrib. Comput.*, 62(2):284–305, 2002.

[Pel05] David Peleg. Distributed Coordination Algorithms for Mobile Robot Swarms: New Directions and Challenges. In Ajit Pal, Ajay D. Kshemkalyani, Rajeev Kumar, and Arobinda Gupta, editors, *Distributed Computing – IWDC 2005*, pages 1–12, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[PV99] Franck Petit and Vincent Villain. Time and Space Optimality of Distributed Depth-First Token Circulation Algorithms. In Yuri Breitbart, Sajal K. Das, Nicola Santoro, and Peter Widmayer, editors, *Distributed Data & Structures 2, Records of the 2nd International Meeting (WDAS 1999)*, volume 6 of *Proceedings in Informatics*, pages 91–106. Carleton Scientific, 1999.

[Ray91] Michel Raynal. A Distributed Solution to the k-out of-M Resources Allocation Problem. In *Advances in Computing and Information - ICCI'91, International Conference on Computing and Information, Ottawa, Canada, May 27-29, 1991, Proceedings*, pages 599–609, 1991.

[RST01] Ronald L. Rivest, Adi Shamir, and Yael Tauman. How to Leak a Secret. In Colin Boyd, editor, *Advances in Cryptology — ASIACRYPT 2001*, pages 552–565, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[San06] Nicola Santoro. *Design and Analysis of Distributed Algorithms (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, USA, 2006.

[Seg83] A Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, IT-29:23–35, 1983.

[SOK+19] Yuichi Sudo, Fukuhito Ooshita, Hirotsugu Kakugawa, Toshimitsu Masuzawa, Ajoy K. Datta, and Lawrence L. Larmore. Loosely-Stabilizing Leader Election for Arbitrary Graphs in Population Protocol Model. *IEEE Trans. Parallel Distrib. Syst.*, 30(6):1359–1373, 2019.

[Tel01] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, New York, NY, USA, 2nd edition, 2001.

[The12] The Coq Development Team. *The Coq Proof Assistant Documentation*, June 2012.

[Tix06] Sébastien Tixeuil. *Toward self-stabilizing large-scale systems*. Habilitation à diriger des recherches, Université Paris Sud - Paris XI, 2006.

[TOKM12] Tomoya Takimoto, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Communication-Efficient Self-stabilization in Wireless Networks. In Andréa W. Richa and Christian Scheideler, editors, *Stabilization, Safety, and Security of Distributed Systems - 14th International Symposium, SSS 2012, Toronto, Canada, October 1-4, 2012. Proceedings*, volume 7596 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2012.

[Var00] George Varghese. Self-Stabilization by Counter Flushing. *SIAM Journal on Computing*, 30(2):486–510, 2000.

[VJ00] George Varghese and Mahesh Jayaram. The fault span of crash failures. *J. ACM*, 47(2):244–293, 2000.

[YK96] Masafumi Yamashita and Tsunehiko Kameda. Computing on Anonymous Networks: Part I-Characterizing the Solvable Cases. *IEEE Transactions on Parallel and Distributed Systems*, 7(1):69–89, 1996.