Weak vs. Self vs. Probabilistic Stabilization

Stéphane Devismes, Sébastien Tixeuil, Masafumi Yamashita

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Weak vs. *Self* vs. *Probabilistic Stabilization*

Stéphane Devismes — Sébastien Tixeuil — Masafumi Yamashita

N° 1

November 2007

Thème NUM

ISSN 0249-6399 ISRN INRIA/RR--1--FR+ENG

Weak *vs.* Self *vs.* Probabilistic Stabilization

Stéphane Devismes* , Sébastien Tixeuil† , Masafumi Yamashita‡

Thème NUM — Systèmes numériques
Projet Grand large

Rapport de recherche n° 1 — November 2007 — 23 pages

Abstract: Self-stabilization is a strong property that guarantees that a network always resume correct behavior starting from an arbitrary initial state. Weaker guarantees have later been introduced to cope with impossibility results: probabilistic stabilization only gives probabilistic convergence to a correct behavior. Also, weak stabilization only gives the possibility of convergence.

In this paper, we investigate the relative power of weak, self, and probabilistic stabilization, with respect to the set of problems that can be solved. We formally prove that in that sense, weak stabilization is strictly stronger that self-stabilization. Also, we refine previous results on weak stabilization to prove that, for practical schedule instances, a deterministic weak-stabilizing protocol can be turned into a probabilistic self-stabilizing one. This latter result hints at more practical use of weak-stabilization, as such algorthms are easier to design and prove than their (probabilistic) self-stabilizing counterparts.

Key-words: Distributed systems, Distributed algorithm, Self-stabilization, Weak-stabilization, Probabilistic self-stabilization

* CNRS, Université Paris-Sud, France
† Université Paris 6, LIP6-CNRS & INRIA, France
‡ CSCE, Kyushu University, Japan

Unité de recherche INRIA Futurs
Parc Club Orsay Université, ZAC des Vignes,
4, rue Jacques Monod, 91893 ORSAY Cedex (France)
Téléphone : +33 1 72 92 59 00 — Télécopie : +33 1 60 19 66 08

Stabilisation faible *vs.* Auto-Stabilisation *vs.* Stabilisation probabiliste

Résumé : L'auto-stabilisation est une propriété forte qui assure qu'un réseau retrouve toujours un comportement correct quel que soit son état initial. Des propriétés plus faibles que l'auto-stabilisation ont été définies pour résoudre des résultats d'impossibilité: l'auto-stabilisation probabiliste garantit uniquement une convergence probabiliste vers un comportement correct; la stabilisation faible garantit simplement une possibilité de convergence à partir de n'importe quel état du système.

Dans cet article, nous nous intéressons aux puissances d'expression relatives de la stabilisation faible, de l'auto-stabilisation déterministe et de l'auto-stabilisation probabiliste. Nous prouvons qu'en pratique la stabilisation faible a réellement un pouvoir d'expression plus fort que l'auto-stabilisation déterministe (*i.e.*, elle permet de résoudre plus de problèmes que l'auto-stabilisation déterministe). Ensuite, nous affinons des résultats antérieurs sur la stabilisation faible pour prouver que du point de vue pratique un protocole faiblement stabilisant déterministe peut être transformé en un protocole auto-stabilisant probabiliste. Ce résultat démontre l'intérêt pratique de la stabilisation faible puisque de tels algorithmes sont plus simples à écrire et à prouver que leurs équivalents auto-stabilisants (probabilistes).

Mots-clés : Systèmes distribués, Algorithme distribué, Auto-stabilisation, Stabilisation faible, Auto-stabilisation probabiliste

1 Introduction

Self-stabilization [10, 11] is a versatile technique to withstand *any* transient fault in a distributed system or network. Informally, a protocol is self-stabilizing if, starting from *any* initial configuration, *every* execution eventually reaches a point from which its behavior is correct. Thus, self-stabilization makes no hypotheses on the nature or extent of faults that could hit the system, and recovers from the effects of those faults in a unified manner.

Such versatility comes with a cost: self-stabilizing protocols can make use of a large amount of resources, may be difficult to design and to prove, or could be unable to solve some fundamental problems in distributed computing. To cope with those issues, several weakened forms of self-stabilization have been investigated in the literature. *Probabilistic self-stabilization* [17] weakens the guarantee on the convergence property: starting from any initial configuration, an execution reaches a point from which its behavior is correct with probability 1. *Pseudo-stabilization* [7] relaxes the notion of "point" in the execution from which the behavior is correct: every execution simply has a suffix that exhibits correct behavior, yet the time before reaching this suffix is unbounded. The notion of *k-stabilization* [2] prohibits some of the configurations from being possible initial states, and assumes that an initial configuration may only be the result of k faults (the number of faults being defined as the number of process memories to change to reach a correct configuration). Finally, the *weak-stabilization* [13] stipulates that starting from *any* initial configuration, *there exists* an execution that eventually reaches a point from which its behavior is correct.

Probabilistic self-stabilization was previously used to reduce resource consumption [15] or to solve problems that are known to be impossible to solve in the classical deterministic setting [14], such as graph coloring, or token passing. Also, it was shown that the well known alternating bit protocol is pseudo-stabilizing, but not self-stabilizing, establishing a strict inclusion between the two concepts. For the case of k-stabilization, [12, 18] shows that if not all possible configurations are admissible as initial ones, several problems that can not be solved in the self-stabilizing setting (*e.g.* token passing) can actually be solved in a k-stabilizing manner. As for weak-stabilization, it was only shown [13] that a sufficient condition on the scheduling hypotheses makes a weak-stabilizing solution self-stabilizing.

From a problem-centric point of view, the probabilistic, pseudo, and k variants of stabilization have been demonstrated strictly more powerfull that classical self-stabilization, in the sense that they can solve problems that are otherwise unsolvable. This comforts the intuition that they provide weaker guarantees with respect to fault recovery. In contrast, no such knowledge is available regarding weak-stabilization.

In this paper, we address the latter open question, and investigate the power of weak-stabilization. Our contribution is twofold: *(i)* we prove that from a problem centric point of view, weak-stabilization is stronger than self-stabilization (both for static problems, such as leader election, and for dynamic problems, such as token passing), and *(ii)* we show that there exists a strong relationship between deterministic weak-stabilizing algorithms and probabilistic self-stabilizing ones. Practically, any deterministic weak-stabilizing protocol can be transformed into a probabilistic self-stabilizing protocol performing under a probabilistic scheduler, as we demonstrate in the sequel of the paper. This results has practical

impact: it is much easier to design and prove a weak-stabilizing solution than a probabilistic one; so if new simple weak-stabilizing solutions appear in the future, our scheme can automatically make them self-stabilizing in the probabilistic sense.

The remaining of the paper is organized as follows. In the next section we present the model we consider in this paper. In Section 3, we propose weak-stabilizing algorithms for problems having no deterministic self-stabilizing solutions. In Section 4, we show that under some scheduling assumptions, a weak-stabilizing system can be seen as a probabilistic self-stabilizing one.

2 Model

Graph Definitions. An *undirected graph* G is a couple (V,E) where V is a set of N *nodes* and E is a set of *edges*, each edge being a pair of distinct nodes. Two nodes p and q are said to be *neighbors iff* $\{p,q\} \in E$. Γ_p denotes the set of p's neighbors. Δ_p denotes the *degree* of p, *i.e.*, $|\Gamma_p|$. By extention, we denote by Δ the degree of G, *i.e.*, $\Delta = \max(\{\Delta_p, p \in V\})$.

A *path* of lenght k is a sequence of nodes $p_0, \ldots, p_k$ such that $\forall i$, $0 \leq i < k$, p_i and p_{i+1} are neighbors. The path $\mathcal{P} = p_0, \ldots, p_k$ is said *elementary* if $\forall i,j$, $0 \leq i < j \leq k$, $p_i \neq p_j$. A path $\mathcal{P} = p_0, \ldots, p_k$ is called *cycle* if $p_0, \ldots, p_{k-1}$ is elementary and $p_0 = p_k$. We call *ring* any graph isomorph to a cycle.

An undirected graph $G = (V,E)$ is said *connected iff* there exists a path in G between each pair of distinct nodes. The *distance* between two nodes p and q in an undirected connected graph $G = (V,E)$ is the length of the smallest path between p and q in G. We denode the distance between p and q by $d(p,q)$. The diameter D of G is equal to $\max(\{d(p,q), p \in V \wedge q \in V\})$. The eccentricity of a node p, noted $ec(p)$, is equal to $\max(\{d(p,q), q \in V\})$. A node p is a *center* of G if $\forall q \in V$, $ec(p) \leq ec(q)$.

We call *tree* any undirected connected acyclic graph. In a tree graph, we distinghish two types of nodes: the *leaves* (*i.e.*, any node p such that $\Gamma_p = 1$) and the *internal nodes* (*i.e.*, any node p such that $\Gamma_p > 1$). Below, we recall a well-known result about the centers in the trees.

Property 1 ([5]) *A tree has a unique center or two neighboring centers.*

Distributed Systems. A *distributed system* is a finite set of communicating state machines called *processes*. We represent the *communication network* of a distributed system by the undirected connected graph $G = (V,E)$ where V is the set of N processes and E is a set of edges such that $\forall p,q \in V$, $\{p,q\} \in E$ *iff* p and q can directly communicate together. Here, we consider *anonymous* distributed systems, *i.e.*, the processes can only differ by their degrees. We assume that each process can distinguish all its neighbors using *local indexes*, these indexes are stored in $Neig_p$. For sake of simplicity, we assume that $Neig_p = \{0, \ldots, \Delta_p - 1\}$. In the following, we will indifferently use the *label* q to designate the process q or the local index of q in the code of some process p.

The communication among neighboring processes is carried out using a *finite* number of *shared variables*. Each process holds its own set of shared variables where it is the only

able to write but where each of its neighbors can read. The *state* of a process is defined by the values of its variables. A *configuration* of the system is an instance of the state of its processes. A process can change its state by executing its *local algorithm*. The local algorithm executed by each process is described by a finite set of guarded actions of the form: $\langle label \rangle :: \langle guard \rangle \rightarrow \langle statement \rangle$. The guard of an action at Process p is a boolean expression involving some variables of p and its neighbors. The statement of an action of p updates some variables of p. An action can be executed only if its guard is satisfied. We assume that the execution of any action is *atomic*. An action of some process p is said enabled in the configuration γ *iff* its guard is *true*. By extention, p is said *enabled* in γ *iff* at least one of its action is enabled in γ.

We model a distributed system as a *transition system* $\mathcal{S} = (\mathcal{C},\mapsto,\mathcal{I})$ where $\mathcal{C}$ is the set of system configuration, $\mapsto$ is a binary transition relation on $\mathcal{C}$, and $\mathcal{I} \subseteq \mathcal{C}$ is the set of initial configurations. An *execution* of $\mathcal{S}$ is a *maximal* sequence of configurations $\gamma_0, \ldots, \gamma_{i-1}, \gamma_i, \ldots$ such that $\gamma_0 \in \mathcal{I}$ and $\forall i > 0$, $\gamma_{i-1} \mapsto \gamma_i$ (in this case, $\gamma_{i-1} \mapsto \gamma_i$ is referred to as a *step*). Any configuration γ is said *terminal* if there is no configuration γ' such that $\gamma \mapsto \gamma'$. We denote by $\gamma \rightsquigarrow \gamma'$ the fact that γ' is *reachable* from γ, *i.e.*, there exists an execution starting from γ and containing γ'.

A *scheduler* is a predicate over the executions. In any execution, each step $\gamma \mapsto \gamma'$ is obtained by the fact that a *non-empty* subset of enabled processes *atomically execute* an action. This subset is chosen according to the scheduler. A scheduler is said *central* [10] if it chooses *one* enabled process to execute an action in any execution step. A scheduler is said *distributed* [6] if it chooses *at least one* enabled process to execute an action in any execution step. A scheduler may also have some *fairness* properties ([11]). A scheduler is *strongly fair* (the strongest fairness assumption) if every process that is enabled *infinitely often* is eventually chosen to execute an action. A scheduler is *weakly fair* if every *continuously* enabled process is eventually chosen to execute an action. Finally, the *proper* scheduler is the weakest fairness assumption: it can forever prevent a process to execute an action except if it is the only enabled process. As the strongly fair scheduler is the strongest fairness assumption, any problem that cannot be solved under this assumption cannot be solved for all fairness assumptions. In contrast, any algorithm working under the proper scheduler also works for all fairness assumptions.

We call `P-variable` any variable v such that there exists a statement of an action where v is randomly assigned. Any variable that is not a `P-variable` is called `D-variable`. Each random assignation of the `P-variable` v is assumed to be performed using a random function $\texttt{Rand}_v$ which returns a value in the domain of v. A system is said *probabilistic* if it contains at least one `P-variable`, otherwise it is said *deterministic*. Let $\mathcal{S} = (\mathcal{C},\mapsto,\mathcal{I})$ be a probabilistic system. Let $\texttt{Enabled}(\gamma)$ be the set of processes that are enabled in $\gamma \in \mathcal{C}$. $\mathcal{S}$ satisfies: for any subset $\texttt{Sub}(\gamma) \subseteq \texttt{Enabled}(\gamma)$, the sum of the probabilities of the execution steps determined by γ and `Sub` is equal to 1.

Stabilizing Systems. Let $\mathcal{S} = (\mathcal{C},\mapsto,\mathcal{I})$ be a system such that $\mathcal{C} = \mathcal{I}$ (*n.b.*, in the following any system $\mathcal{S} = (\mathcal{C},\mapsto,\mathcal{I})$ such that $\mathcal{C} = \mathcal{I}$ will be simply denoted by $\mathcal{S} = (\mathcal{C},\mapsto)$). Let $\mathcal{SP}$ be a specification, *i.e.*, a particular predicate defined over the executions of $\mathcal{S}$.

Definition 1 (Deterministic Self-Stabilization [10]) *$\mathcal{S}$ is* deterministically self-stabilizing *for $\mathcal{SP}$ if there exists a non-empty subset of $\mathcal{C}$, noted $\mathcal{L}$, such that:* (i) *Any execution of $\mathcal{S}$ starting from a configuration of $\mathcal{L}$ always satisfies $\mathcal{SP}$ (*Strong Closure Property*), and* (ii) *Starting from any configuration,* any execution *of $\mathcal{S}$ reaches in a finite time a configuration of $\mathcal{L}$ (*Certain Convergence Property*).*

Definition 2 (Probabilistic Self-Stabilization [17]) *$\mathcal{S}$ is* probabilistically self-stabilizing *for $\mathcal{SP}$ if there exists a non-empty subset of $\mathcal{C}$, noted $\mathcal{L}$, such that:* (i) *Any execution of $\mathcal{S}$ starting from a configuration of $\mathcal{L}$ always satisfies $\mathcal{SP}$ (*Strong Closure Property*), and* (ii) *Starting from any configuration, any execution of $\mathcal{S}$ reaches a configuration of $\mathcal{L}$* with Probability 1 *(*Probabilistic Convergence Property*).*

Definition 3 (Deterministic Weak-Stabilization [13]) *$\mathcal{S}$ is* deterministically weak-stabilizing *for $\mathcal{SP}$ if there exists a non-empty subset of $\mathcal{C}$, noted $\mathcal{L}$, such that:* (i) *Any execution of $\mathcal{S}$ starting from a configuration of $\mathcal{L}$ always satisfies $\mathcal{SP}$ (*Strong Closure Property*), and* (ii) *Starting from any configuration,* there always exists an execution *that reaches a configuration of $\mathcal{L}$ (*Possible Convergence Property*).*

Note that the configurations from which $\mathcal{S}$ always satisfies $\mathcal{SP}$ ($\mathcal{L}$) are called *legitimate configurations*. Conversely, every configuration that is not legitimate is *illegitimate*.

3 From Self to Weak Stabilization

In this section, we exhibit two problems that can not be solved by a deterministic self-stabilizing protocol, yet admit surprisingly simple deterministic weak-stabilizing ones. Thus, from a problem-centric point of view, weak-stabilization is stronger than self-stabilization. This result is mainly due to the fact that a given scheduler is appreciated differently when we consider self or weak stabilization. In the self-stabilizing setting, the scheduler is seen as an *adversary*: the algorithm must work properly despite the "bad behavior" of the scheduler. Indeed, it is sufficient to exhibit an execution that satisfies the scheduler predicate yet prevents the algorithm from converging to a legitimate configuration to prove the absence of self-stabilization. Conversely, in weak-stabilization, the scheduler can be viewed as a *friend*: to prove the property of weak-stabilization, it is sufficient to show that, for any configuration γ, there exists an execution starting from γ that satisfies the scheduler predicate and converges. As a matter of fact, the effect of the scheduler is reversed in weak and self stabilization: the strongest the scheduler is (*i.e.* the more executions are included in the scheduler predicate), the easier the weak-stabilization can be established, but the harder self-stabilization is.

When the scheduler is *synchronous* [16] (*i.e.*, a scheduler that chooses *every* enabled process at each execution step) the notions of deterministic weak-stabilization and deterministic self-stabilization are equivalent, as proved in the following.

Theorem 1 *Under a synchronous scheduler, an algorithm is deterministically weak-stabilizing* iff *it is also deterministically self-stabilizing.*

Proof.

If. Consider algorithm $\mathcal{P}$ that is deterministically weak-stabilizing under a synchronous scheduler. First, $\mathcal{P}$ satisfies the *strong closure* property. It remains then to show that $\mathcal{P}$ satisfies the *certain convergence* property.

By Definition 3, starting from any configuration γ, there exists an execution of $\mathcal{P}$ that converges to a legitimate configuration. Now, under a synchronous scheduler, there is an unique execution starting from γ because $\mathcal{P}$ is deterministic. Hence, $\mathcal{P}$ trivially satisfies the following assertion "starting from any configuration, any execution of $\mathcal{P}$ converges to a legitimate configuration under a synchronous scheduler" (the *certain convergence* property).

Only If. By Definition, any deterministic self-stabilizing algorithm is also a deterministic weak-stabilizing algorithm under the same scheduler. □

We now exhibit two examples of problems that admit weak-stabilizing solutions but no self-stabilizing ones: the token passing and the leader election.

3.1 Token Circulation

In this subsection, we consider the problem of Token Circulation in a unidirectional ring, with a strongly fair distributed scheduler. This problem is one of the most studied problems in self-stabilization, and is often regarded as a "benchmark" for new algorithms and concepts. The consistent direction is given by a constant local pointer $Pred$: for any process p, $Pred_p$ designates a neighbor q as the *predecessor* (resp. p is the *successor* of q) in such way that q is the predecessor of p *iff* p is not the predecessor of q.

Definition 4 (Token Circulation) *The* token circulation *problem consists in circulating a single token in the network in such way that every process holds the token infinitely often.*

In [16], Herman shows, using a previous result of Angluin [1], that the deterministic self-stabilizing token circulation is impossible in anonymous networks because there is no ability to break symmetry. We now show that, contrary to deterministic self-stabilization, deterministic weak-stabilizing token circulation under distributed strongly fair scheduler exists in an anonymous unidirectional ring.

Our starting point is the $(N-1)$-fair algorithm of Beauquier *et al.* proposed in [3] (presented as Algorithm 1). We show that Algorithm 1 is actually a deterministic weak-stabilizing token circulation protocol. Roughly speaking, $(N-1)$-fairness implies that in any execution, *(i)* every process p performs actions infinitely often, and *(ii)* between any two actions of p, any other process executes at most $N-1$ actions. The memory requirement of Algorithm 1 is $\log(m_N)$ bits per process where m_N is the smallest integer not dividing N (the ring size). Note that it is also shown in [3] that this memory requirement is minimal to obtain any probabilistic self-stabilizing token circulation under a distributed scheduler (such a probabilistic self-stabilizing token circulation can be found in [9]).

A process p maintains a single counter variable: dt_p such that $dt_p \in [0 \dots m_N - 1]$. This variable allows p to know if it holds the token or not. Actually, a process p holds a token

Algorithm 1 Code for every process p

Variable: $dt_p \in [0 \dots m_N - 1]$
Macro:
$PassToken_p \quad = \quad dt_p \leftarrow (dt_{Pred_p} + 1) \bmod m_N$
Predicate:
$Token(p) \quad \equiv \quad [dt_p \neq ((dt_{Pred_p} + 1) \bmod m_N)]$
Action:
`A` :: $Token(p) \quad \rightarrow \quad PassToken_p$

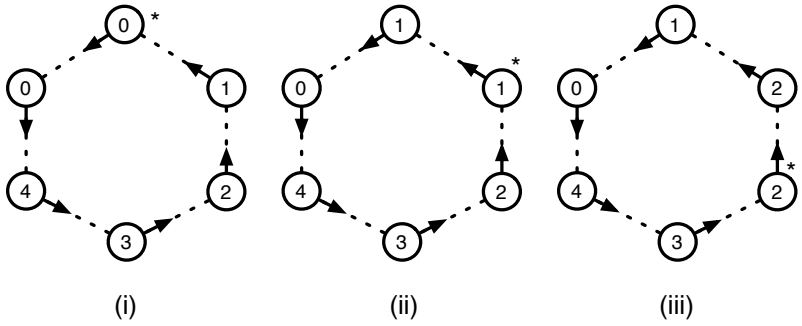

Figure 1: Example of an execution starting from a legitimate configuration.

iff $dt_p \neq ((dt_{Pred_p} + 1) \bmod m_N)$, *i.e.*, *iff* p satisfies $Token(p)$. In this case, Action `A` is enabled at p. This action allows p to pass the token to its *successor*.

Figure 1 depicts an execution of Algorithm 1 starting from a legitimate configuration, *i.e.*, a configuration where there is exactly one process that satisfies Predicate $Token$. In the figure, the outgoing arrows represent the $Pred$ pointers and the integers represent the dt values. In this example, the ring size N is equal to 6. So, $m_N = 4$. In each configuration, the only process with an asterisk is the only token holder: by executing Action `A`, it passes the token to its successor.

Theorem 2 *Algorithm 1 is a* deterministic weak-stabilizing token passing *algorithm under a distributed strongly fair scheduler.*

Proof. Given in the appendix (Section A, page 19). □

3.2 Leader Election

In this subsection, we consider anonymous tree-shaped networks and a distributed strongly fair scheduler.

Definition 5 (Leader Election) *The* leader election *problem consists in distinguishing a unique process in the network.*

We first prove that the leader election problem is impossible to solve in our setting in a self-stabilizing way.

Theorem 3 *Assuming a distributed strongly fair scheduler, there is no deterministic self-stabilizing leader election algorithm in anonymous trees.*

Proof. Consider a chain of four processes P_1, P_2, P_3, P_4 (a particular case of tree) and a synchronous execution (a possible behavior of a distributed strongly fair scheduler). Let us denote by $\langle S_1,S_2,S_3,S_4\rangle$ any configuration of the system we consider where S_i ($i \in [1 \dots 4]$) represents the local state of P_i. Let $\mathcal{X}$ be the subset of configurations such that $S_1 = S_4$ and $S_2 = S_3$ (note that $S_1 = S_2 = S_3 = S_4$ is a particular case of such configurations). Of course, in any configuration of $\mathcal{X}$, we cannot distinghish any leader. We now show that $\mathcal{X}$ is closed in a synchronous execution, which proves the impossibility of the deterministic self-stabilizing leader election.

Consider a configuration $\gamma = \langle a,b,b,a\rangle$ of the set $\mathcal{X}$. As we cannot distinghish any leader in γ, γ must not be terminal. So, consider an arbitrary execution starting from γ and let γ' be the configuration that follows γ in the execution. The three following cases are possible for the step $\gamma \mapsto \gamma'$:

- Only P_1 and P_4 are enabled in γ. As the system is deterministic and the execution is synchronous, there only one possible step: P_1 and P_4 changes their local state in the same deterministical way. So, S_1 is still identical to S_4 in γ', *i.e.*, $\gamma' = \langle a,b',b',a\rangle$.

- Only P_2 and P_3 are enabled in γ. As the system is deterministic and the execution is synchronous, there only one possible step: P_2 and P_3 changes their local state in the same deterministical way. So, S_2 is still identical to S_3 in γ', *i.e.*, $\gamma' = \langle a,b',b',a\rangle$.

- All processes are enabled. In this case, we trivially have $\gamma' = \langle a',b',b',a'\rangle$.

Hence, $\gamma' \in \mathcal{X}$, which proves that $\mathcal{X}$ is closed. □

We now provide two weak-stabilizing solutions for the same problem in the same setting, with different space complexities. Both solutions are more intuitive and simpler to design than self-stabilizing ones in slightly different settings.

A solution using $\log N$ bits. A straighforward solution is to use the algorithm provided in [4]. This algorithm uses $\log N$ bits and finds the centers of a tree network: starting from any configuration, the system reaches in a finite time a terminal configuration where any process p satisfies a particular local predicate $Center(p)$ *iff* p is a center of the tree. From Property 1, two cases are then possible in a terminal configuration: either a unique process satisfies $Center$ or two neighboring processes satisfy $Center$.

If there is only one process p satisfying $Center(p)$, it is considered as the leader.

Now, assume that there are two neighboring processes p and q that satisfy $Center$. In this case, p (resp. q) is able to locally detect that q (resp. p) is the other center (see [4] for details). So, we use an additional boolean B to break the tie. If $B_p \neq B_q$, then the only

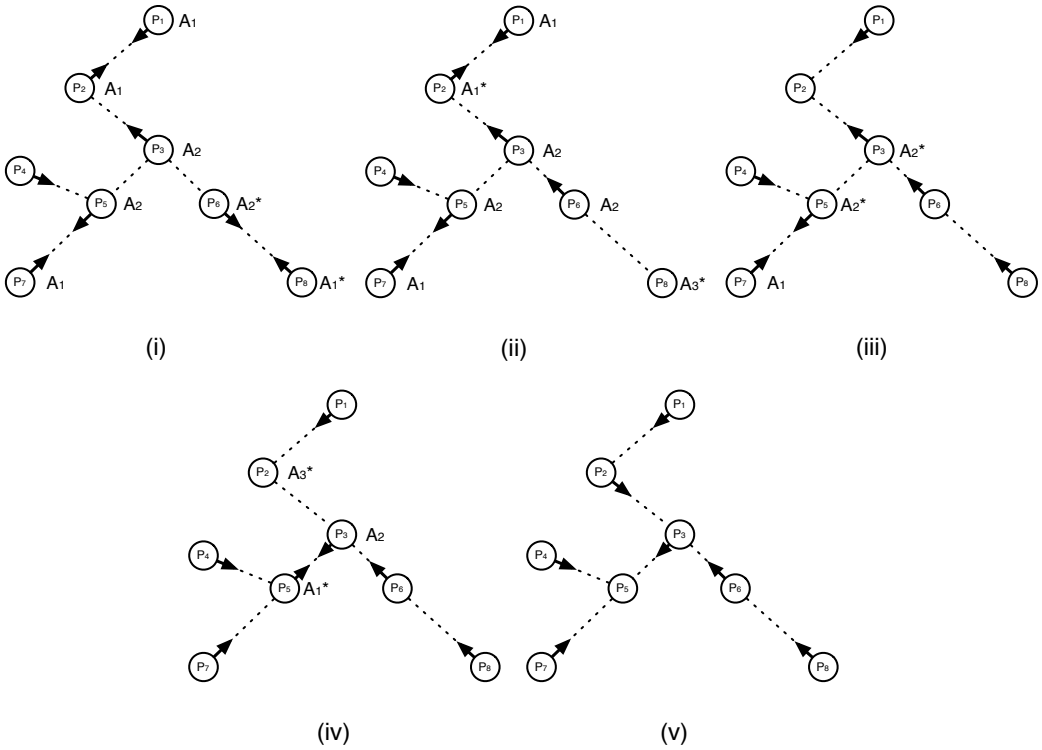

Figure 2: Example of possible convergence.

center satisfying $B = true$ is considered as the leader. Otherwise, both p and q are enabled to execute $B \leftarrow \neg B$. So, from any configuration where the two centers have been found but no leader is distinguished, this is always possible to reach a terminal configuration where a leader is distinghished in one step: if only one of the two centers moves.

Another solution using $\log \Delta$ bits. In this solution (Algorithm 2), each process p maintains a single variable: Par_p such that $Par_p \in Neig_p \cup \{\perp\}$. p considers itself as the leader *iff* $Par_p = \perp$. If $Par_p \neq \perp$, the *parent* of p is the neighbor pointed out by Par_p, conversely p is said to be a *child* of this process.

Algorithm 2 tries to reach a terminal configuration where: *(i)* exactly one process l is designated as the leader, and *(ii)* all other processes q point out using Par_q their neighbor that is the closest from l. In other words, Algorithm 2 computes an arbitrary orientation of the network in a deterministic weak-stabilizing manner.

Algorithm 2 uses the following strategy:

Algorithm 2 Code for any process p

Variable: $Par_p \in Neig_p \cup \{\perp\}$
Macro:
$Children_p \quad = \quad \{q \in Neig_p,\ Par_q = p\}$
Predicates:
$isLeader(p) \quad \equiv \quad (Par_p = \perp)$
Actions:

$\mathtt{A_1}$	::	$(Par_p \neq \perp) \wedge (\lvert Children_p \rvert = \lvert Neig_p \rvert)$	$\rightarrow$	$Par_p \leftarrow \perp$
$\mathtt{A_2}$	::	$(Par_p \neq \perp) \wedge [Neig_p \setminus (Children_p \cup \{Par_p\}) \neq \emptyset]$	$\rightarrow$	$Par_p \leftarrow (Par_p + 1) \bmod \Delta_p$
$\mathtt{A_3}$	::	$(Par_p = \perp) \wedge (\lvert Children_p \rvert < \lvert Neig_p \rvert)$	$\rightarrow$	$Par_p \leftarrow \min_{\prec_p}(Neig_p \setminus Children_p)$

1. If a process p such that $Par_p \neq \perp$ is pointed out by all its neighbors, then this means that all its neighbors consider it as the leader. As a consequence, p sets Par_p to $\perp$ (Action $\mathtt{A_1}$), *i.e.*, it starts to consider itself as the leader.

2. If a process p such that $Par_p \neq \perp$ has a neighbor which is neither its parent nor one of its children, then this means that not all processes among p and its neighbors consider the same process as the leader. In this case, p changes its parent by simply incrementing its parent pointer modulus Δ_p (Action $\mathtt{A_2}$). Hence, from any configuration, it is always possible that all processes satisfying $Par \neq \perp$ eventually agree on the same leader.

3. Finally, if a process p satisfies $Par_p = \perp$ and at least one of neighbor q does not satisfy $Par_q = p$, then this means that q considers another process as the leader. As a consequence, p stops to consider itself as the leader by pointing out one of its non-child neighbor (Action $\mathtt{A_3}$).

Figure 2 depicts an example of execution of Algorithm 2 that converges. In the figure, the circles represent the processes and the dashed lines correspond to the neighboring relations. The labels of processes are just used for the ease of explanation. Then, if there is an arrow outgoing from process P_i, this arrow designates the neighbor pointed out by Par_{P_i}. In contrast, $Par_{P_i} = \perp$ holds if there is no arrow outgoing from process P_i. Any label A_j beside a process P_i means that Action $\mathtt{A_j}$ is enabled at P_i. Finally, some labels A_j are sometime asterisked meaning that their corresponding actions is executed in the next step.

In initial configuration (i), no process satisfies $Par = \perp$, *i.e.*, no process consider itself as the leader. However, P_1, P_2, P_7, and P_8 are pointed out by all their respective neighbors. So, these processes are candidates to become the leader (Action $\mathtt{A_1}$). Also, note that P_3, P_5, and P_6 are enabled to execute Action $\mathtt{A_2}$: they have a neighbor that is neither their parent or one of their children. Finally, note that P_4 is in a stable local state. In the first step $(i) \mapsto (ii)$, P_6 and P_8 execute their enabled action: in (ii), there is a unique leader (P_8) but it has no child, *i.e.*, no other process agrees on its leadership. So P_8 is enabled to lose its leadership (Action $\mathtt{A_3}$). In $(ii) \mapsto (iii)$, P_8 looses its leadership (Action $\mathtt{A_3}$) but P_2 becomes a leader (Action $\mathtt{A_1}$). So, there is still a unique leader (P_2) in the configuration (iii). In the step $(iii) \mapsto (iv)$, P_3 and P_5 change their parent to P_5 and P_3, respectively. As a consequence, Action $\mathtt{A_1}$ becomes enabled at P_5 in (iv). However, P_2 is also enabled in (iv) to lose its leadership (Action $\mathtt{A_3}$). In $(iv) \mapsto (v)$, P_2 and P_5 execute their respective enabled action and the system reach the terminal configuration (v).

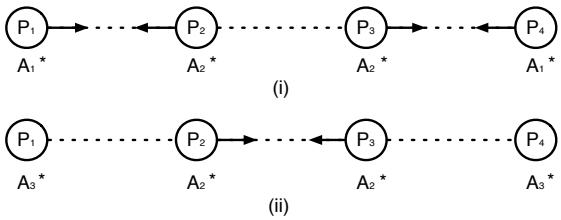

Figure 3: Example of an execution that does not converge.

Figure 3 illustrates the fact that Algorithm 2 is deterministically weak-stabilizing but not deterministically self-stabilizing under a distributed scheduler (for all fairness assumptions). Actually Figure 3 show that there is some infinite executions of Algorithm 2 that never converge. This example is quite simple: starting from the configuration (i), if the execution is synchronous, the system reaches configuration (ii) in one step, then we retreive configuration (i) after two steps, and so on. This sequence can be repeated indefinitely. So, there is a possible execution starting from (i) that never converges.

Theorem 4 *Algorithm 2 is a* deterministic weak-stabilizing leader election *algorithm under a distributed strongly fair scheduler.*

Proof. Given in the appendix (Section B, page 20). □

4 From Weak to Probabilistic Stabilization

In [13], Gouda shows that deterministic weak-stabilization is a "good approximation" of deterministic self-stabilization[1] by proving the following theorem:

Theorem 5 ([13]) *Any deterministic weak-stabilizing system is also a deterministic self-stabilizing system if:*

- *The system has a finite number of configurations, and*
- *Every execution satisfies the* Gouda's strong fairness *assumption where* Gouda's strong fairness *means that, for every transition $\gamma \mapsto \gamma'$, if γ occurs infinitely often in an execution e, then $\gamma \mapsto \gamma'$ also appears infinitely often in e.*

From Theorem 5, one may conclude that deterministic weak-stabilization and deterministic self-stabilization are equivalent under the distributed strongly fair scheduler. This would contradict the results presented in Section 3. Actually, this is not the case: we prove

[1] This result has been proven for the central scheduler but it is easy to see that the proof also holds for any scheduler.

in Theorem 6 that the Gouda's strong fairness assumption is (strictly) stronger than the classical notion of strong fairness. A less ambiguous and more practical characterization of deterministic weak-stabilization is the following: under Gouda's strong fairness assumption, the scheduler does not behave as an adversary but rather as a probabilistic one (*i.e.*, a deterministic weak-stabilizing system may never converge but if it is lucky, it converges). Hence, under a distributed randomized scheduler [8], which chooses among enabled processes with a (possibly) uniform probability which are activated, any weak-stabilizing system converges with probability 1 despite an arbitrary initial configuration (Theorem 7).

Theorem 6 *The* Gouda's strong fairness *is stronger than the* strong fairness.

Proof. As Algorithm 1 (page 8) is a deterministic weak-stabilizing token circulation with a finite number of configurations, it is also a deterministic self-stabilizing token circulation under the *Gouda's strongly fairness assumption* (Theorem 5). We now show the lemma by exhibiting an execution of Algorithm 1 that does not converge under the central strongly fair scheduler (a similar counter-example can be also derived for a synchronous scheduler).

Consider a ring of six processes $p_0, \ldots, p_5$. Consider a configuration γ_0 where only p_0 and p_3 hold a token. Both p_0 and p_3 are enabled in γ_0. Assume that only p_0 passes its token in the step $\gamma_0 \mapsto \gamma_1$. In γ_1, p_1 and p_3 hold a token. Assume now that only p_3 passes its token in the step $\gamma_1 \mapsto \gamma_3$ and so on. It is straightforward that if the two tokens alternatively move at each step, then the execution never converges despite it respects the central strongly fair scheduler. □

We now show that the *randomized scheduler* defined below is a notion that is, in some sense, equivalent to the *Gouda's strong fairness*.

Definition 6 (Randomized Scheduler [8]) *A scheduler is said* randomized *if it randomly chooses with a uniform probability the enabled processes that execute an action in each step.*

Note that under a *central randomized scheduler*, in every step the unique process that executes an action is chosen with a uniform probability among the enabled processes. Similarly, under a *distributed randomized scheduler*, in every step the processes (at least one) that executes an action are chosen with a uniform probability among the enabled processes.

Theorem 7 *Let $\mathcal{P}$ be a deterministic algorithm having a finite number of configurations. $\mathcal{P}$ is deterministically self-stabilizing under the Gouda's fairness assumption* iff *$\mathcal{P}$ is probabilistically self-stabilizing under a randomized scheduler.*

Proof. Let $\mathcal{P}$ be a deterministic algorithm having a finite number of configurations.

If. Assume that $\mathcal{P}$ is deterministically self-stabilizing under the Gouda's fairness assumption. First, $\mathcal{P}$ satisfies the *strong closure* property. Hence, it remains to show that $\mathcal{P}$ also satisfies the *probabilistic convergence* property.

Assume, by the contradiction, that there exists an execution e of $\mathcal{P}$ that do not converge with a probability 1 under a distributed randomized scheduler. As the number of configurations of $\mathcal{P}$ is finite, there exists at least one configuration γ_0 that occurs infinitely often in e. Then, as $\mathcal{P}$ is deterministically self-stabilizing under the *Gouda's fairness assumption*, there exists an execution $\gamma_0, \gamma_1, \ldots, \gamma_k$ such that γ_k is a legitimate configuration. Now, as the scheduler is randomized, there is a strictly positive probability that $\gamma_0 \mapsto \gamma_1$ occurs starting from γ_0. Hence, $\gamma_0 \mapsto \gamma_1$ occurs with a probability 1 after a finite number of occurences of γ_0 in e and, as a consequence, γ_1 occurs infinitely often (with the probability 1) in e. Inductively, it is then straightforward that $\forall i \in [1 \ldots k]$, γ_i occurs infinitely often in e with the probability 1. Hence, the legitimate configuration γ_k eventually occurs in e with the probability 1, a contradiction.

Only If. Assume that $\mathcal{P}$ is probabilistically self-stabilizing under a distributed randomized scheduler. First, $\mathcal{P}$ satisfies the *strong closure* property. Then, starting from any configuration, there exists at least one execution that converges to a legitimate configuration: $\mathcal{P}$ satisfies the *possible convergence* property. Hence, $\mathcal{P}$ is *weak-stabilizing* and, by Theorem 5, $\mathcal{P}$ is deterministically self-stabilizing under the Gouda's fairness assumption. □

Theorem 7 claims that if the distributed scheduler does not behave as an adversary, then any deterministic weak-stabilizing system stabilizes with a probability 1. So, we could expect that under a synchronous scheduler, which corresponds to a "friendly" behavior of the distributed scheduler, any weak-stabilizing system also stabilizes. Unfortunately, this is not the case: for example, Figure 3 (page 12) depicts a possible synchronous execution of Algorithm 2 that never converges. In contrast, it is easy to see that under a central randomized scheduler, Algorithms 1 and 2 are still probabilistically self-stabilizing (to prove the weak-stabilization of Algorithms 1 and 2 under a distributed scheduler we never use the fact that more that one process can be activated at each step). Hence, this means that in some cases, the asynchrony of the system helps its stabilization while the synchrony can be pathological. This could seem unintuitive at first, but this is simply due to the fact that a synchronous scheduler maintains symmetry in the system. However, it is desirable to have a solution that works with both a distributed randomized scheduler and a synchronous one. This is the focus of the following paragraph.

Breaking Synchrony-induced Symetry. We now propose a simple transformer that permits to break the symetries when the system is synchronous while keeping the convergence property of the algorithm under a distributed randomized scheduler. Our transformation method consists in simulating a randomized distributed scheduler when the system behaves in a synchronous way (this method was used in the conflict manager provided in [14]): each time an enabled process is activated by the scheduler, it first tosses a coin and then performs the expected action only if the toss returns true.

In our scheme, we add a new boolean random variable B_i in the code of each processor i. We then transform any action $\mathtt{A} :: Guard_{\mathtt{A}} \rightarrow \mathtt{S_A}$ of the input (deterministic weak-stabilizing) algorithm into the following action $\mathtt{Trans(A)}$:

$$\mathtt{Trans(A)} :: Guard_{\mathtt{A}} \rightarrow B_i \leftarrow \mathtt{Rand}_i(true, false); \textbf{ if } B_i \textbf{ then } \mathtt{S_A}$$

Of course, our method does not absolutely forbid synchronous behavior of the system: at any step, there is a strictly positive probability that every enabled process is activated and wins the toss. Such a property is very important because some deterministic weak-stabilizing algorithms under a distributed scheduler require some "synchronous" steps to converge. Such an exemple is provided below.

Consider a network consisting of two neighboring processes, p and q, having a boolean variable B and executing the following algorithm:

Algorithm 3 Code for a process i

Input: j: the neighbor of i
Variable: B_i: boolean
Actions:

$\mathtt{A}_1$	$(\neg B_i \wedge \neg B_j)$	$\rightarrow$	$B_i \leftarrow true$
$\mathtt{A}_2$	$(B_i \wedge \neg B_j)$	$\rightarrow$	$B_i \leftarrow false$

Trivially, Algorithm 3 is deterministically weak-stabilizing under a distributed strongly fair scheduler for the following predicate: $(B_p \wedge B_q)$. Indeed, if $(B_p,B_q) = (true,false)$ or $(false,true)$, then in the next configuration, $(B_p,B_q) = (false,false)$ and from such a configuration, three cases are possible in the next step: *(i)* only $B_p \leftarrow true$, *(ii)* only $B_q \leftarrow true$, or *(iii)* $(B_p,B_q) \leftarrow (true,true)$. In the two first cases, the system retreives a configuration where $(B_p,B_q) = (true,false)$ or $(false,true)$. In the latter case, the system reaches a terminal configuration where $(B_p \wedge B_q)$ holds. Hence, Algorithm 3 requires to converge that p and q move simultaneously when $(B_p,B_q) = (false,false)$. The transformed version of Algorithm 3 trivially converges with the probability 1 under a distributed randomized scheduler as well as a synchronous one because while the system is not in a terminal configuration, the system regulary passes by the configuration $(B_p,B_q) = (false,false)$ and from such a configuration, there is a strictly positive probability that both p and q executes $B \leftarrow true$ in the next step.

Transformer Correctness. Below we prove that our method transforms any deterministic weak-stabilizing system for a distributed scheduler with a finite number of configurations into a randomized self-stabilizing system for a synchronous scheduler. The proof that the transformed system remains a probabilistically self-stabilizing under a randomized scheduler is (trivially) similar and is omitted from the presentation.

Let $\mathcal{S}_{\mathtt{Det}} = (\mathcal{C}_{\mathtt{Det}},\mapsto_{\mathtt{Det}})$ be a system that is deterministically weak-stabilizing for the specification $\mathcal{SP}$ under a distributed scheduler and having a finite number of configurations. Let $\mathcal{L}_{\mathtt{Det}} \subseteq \mathcal{C}_{\mathtt{Det}}$ be the (non-empty) set of legitimate configurations of $\mathcal{S}_{\mathtt{Det}}$. Let $\mathcal{S}_{\mathtt{Prob}} = (\mathcal{C}_{\mathtt{Prob}},\mapsto_{\mathtt{Prob}})$ be the probabilistic system obtained by transforming $\mathcal{S}_{\mathtt{Det}}$ according to the above presented method. By construction, any variable v of $\mathcal{S}_{\mathtt{Det}}$ also exists in $\mathcal{S}_{\mathtt{Prob}}$. So, let us denote by $\gamma_{|\mathcal{S}_{\mathtt{Det}}}$ the projection of the configuration $\gamma \in \mathcal{C}_{\mathtt{Prob}}$ on the variables of $\mathcal{S}_{\mathtt{Det}}$. By Definition, $\forall \gamma \in \mathcal{C}_{\mathtt{Prob}}$, $\gamma_{|\mathcal{S}_{\mathtt{Det}}} \in \mathcal{C}_{\mathtt{Det}}$ and $\forall \alpha \in \mathcal{C}_{\mathtt{Det}}$, $\exists \gamma \in \mathcal{C}_{\mathtt{Prob}}$ such that $\gamma_{|\mathcal{S}_{\mathtt{Det}}} = \alpha$.

Definition 7 *Let* $\mathcal{L}_{\mathtt{Prob}} = \{\gamma \in \mathcal{C}_{\mathtt{Prob}} : \gamma_{|\mathcal{S}_{\mathtt{Det}}} \in \mathcal{L}_{\mathtt{Det}}\}$.

Lemma 1 (Strong Closure) *Any synchronous execution of* $\mathcal{S}_{\mathtt{Prob}}$ *starting from a configuration of* $\mathcal{L}_{\mathtt{Prob}}$ *always satisfies* $\mathcal{SP}$.

Proof. By Definition, $(\mathcal{L}_{\texttt{Prob}} \neq \emptyset)$ and $(\forall \gamma \in \mathcal{L}_{\texttt{Prob}}, \gamma_{|\mathcal{S}_{\texttt{Det}}} \in \mathcal{L}_{\texttt{Det}})$, *i.e.*, the projection of any configuration of $\mathcal{L}_{\texttt{Prob}}$ on the variables of $\mathcal{S}_{\texttt{Det}}$ is a legitimate configuration of $\mathcal{S}_{\texttt{Det}}$. So, it remains to show that any configuration $\gamma \in \mathcal{L}_{\texttt{Prob}}$ satisfies the predicate $\texttt{P} \equiv (\forall \gamma' \in \mathcal{C}_{\texttt{Prob}} : \gamma \mapsto_{\texttt{Prob}} \gamma', \gamma' \in \mathcal{L}_{\texttt{Prob}})$.

Consider any configuration $\gamma \in \mathcal{L}_{\texttt{Prob}}$.

- If γ is a *terminal configuration* (*i.e.*, there is no configuration $\gamma' \in \mathcal{C}_{\texttt{Prob}}$ such that $\gamma \mapsto_{\texttt{Prob}} \gamma'$), then γ trivially satisfies P.
- Assume now that $(\exists \gamma' \in \mathcal{C}_{\texttt{Prob}} : \gamma \mapsto_{\texttt{Prob}} \gamma')$. Consider then any transition $\gamma \mapsto_{\texttt{Prob}} \gamma'$. In this transition, every enabled process p executes its enabled action $\texttt{Trans}_p(\texttt{A})$ (the execution is synchronous). First, any p tosses a coin ($B_p \leftarrow \texttt{Rand}_p(true,false)$). Then, two cases are possible:
  - *If every process p looses the toss* (*i.e.*, $\texttt{Rand}_p(true,false)$ returns *true* for any p), then no assignment is performed on the variables that are commun to $\mathcal{S}_{\texttt{Prob}}$ and $\mathcal{S}_{\texttt{Det}}$. As a consequence, $\gamma'_{|\mathcal{S}_{\texttt{Det}}} = \gamma_{|\mathcal{S}_{\texttt{Det}}}$and, trivially, we have $\gamma' \in \mathcal{L}_{\texttt{Prob}}$.
  - *If some processes win the toss*, then we can remark that any assignment of a variable commun to $\mathcal{S}_{\texttt{Prob}}$ and $\mathcal{S}_{\texttt{Det}}$ performed by Action $\texttt{Trans}_p(\texttt{A})$ exists in Action $\texttt{A}_p$. Now, $\mathcal{S}_{\texttt{Det}}$ satisfies the strong closure property for the set $\mathcal{L}_{\texttt{Det}}$ under a distributed scheduler. So, $\gamma'_{|\mathcal{S}_{\texttt{Det}}} \in \mathcal{L}_{\texttt{Det}}$, *i.e.*, $\gamma' \in \mathcal{L}_{\texttt{Prob}}$.

  Hence, for any transition $\gamma \mapsto_{\texttt{Prob}} \gamma'$, we have $(\gamma \in \mathcal{L}_{\texttt{Prob}}) \Rightarrow (\gamma' \in \mathcal{L}_{\texttt{Prob}})$, *i.e.*, γ satisfies P.

□

As we assume that $\mathcal{C}_{\texttt{Det}}$ is finite and the variables of $\mathcal{S}_{\texttt{Prob}}$ and $\mathcal{S}_{\texttt{Det}}$ differ by just a boolean, the following observation is obvious:

Observation 1 *$\mathcal{C}_{\texttt{Prob}}$ is a finite set.*

Lemma 2 *$\forall \gamma \in \mathcal{C}_{\texttt{Prob}}$, $\exists \gamma' \in \mathcal{L}_{\texttt{Prob}}$, $\gamma \rightsquigarrow \gamma'$ under a synchronous scheduler.*

Proof. Let $\gamma_0 \in \mathcal{C}_{\texttt{Prob}}$. Consider the configuration α_0 such that $\gamma_{0|\mathcal{S}_{\texttt{Det}}} = \alpha_0$. By Definition, there exists an execution of $\mathcal{S}_{\texttt{Det}}$: $\alpha_0, \ldots, \alpha_k$ such that $\alpha_k \in \mathcal{L}_{\texttt{Det}}$. Now, for any execution $\alpha_0, \ldots, \alpha_k$ of $\mathcal{S}_{\texttt{Det}}$ there exists a corresponding execution of $\mathcal{S}_{\texttt{Prob}}$: $\gamma_0, \ldots, \gamma_k$ such that $\forall i \in [1 \ldots k]$, $\gamma_{i|\mathcal{S}_{\texttt{Det}}} = \alpha_i$. Indeed:

(1) The set of enabled processes is the same in α_{i-1} and γ_{i-1}, and

(2) Any step $\gamma_{i-1} \mapsto \gamma_i$ is performed if the subset of enabled processes that win the toss during $\gamma_{i-1} \mapsto \gamma_i$ is exactly the subset of enabled processes that are chosen by the distributed scheduler in $\alpha_{i-1} \mapsto \alpha_i$.

Since, $\gamma_{k|\mathcal{S}_{\texttt{Det}}} = \alpha_k$ and $\alpha_k \in \mathcal{L}_{\texttt{Det}}$, we have $\gamma_k \in \mathcal{L}_{\texttt{Prob}}$ and the lemma is proven. □

Lemma 3 (Probabilistic Convergence) *Starting from any configuration, any synchronous execution of $\mathcal{S}_{\texttt{Prob}}$ reaches a configuration of $\mathcal{L}_{\texttt{Prob}}$ with the probability 1.*

Proof. Consider, by the contradiction, that there exists an execution e of $\mathcal{S}_{\texttt{Prob}}$ that do not reach any a configuration of $\mathcal{L}_{\texttt{Prob}}$ with the probability 1. Then, by Lemma 2, while the system is not in a legitimate configuration it is not in a terminal configuration and, as a consequence, e is infinite. Moreover, as the number of possible configurations of the system is finite (Observation 1), there is a subset of configurations $\texttt{W} \subset \mathcal{C}_{\texttt{Prob}} \setminus \mathcal{L}_{\texttt{Prob}}$ that appears infinitely often in e. By Lemma 2 again, there is two configuration $\gamma \in \texttt{W}$ and $\gamma' \in \mathcal{C}_{\texttt{Prob}} \setminus \texttt{W}$ such that $\gamma \mapsto_{\texttt{Prob}} \gamma'$ but the step $\gamma \mapsto_{\texttt{Prob}} \gamma'$ never appears in e. As execution is synchronous, every enabled process executes an action from γ and depending on the tosses, there is a strictly positive probability that the step $\gamma \mapsto_{\texttt{Prob}} \gamma'$ occurs from γ. Now, as γ appears infinitely often in e, the step $\gamma \mapsto_{\texttt{Prob}} \gamma'$ is performed after a finite number of occurences of γ in e with the probability 1, a contradiction. □

By Lemmas 1 and 3, we get:

Theorem 8 *Assuming a synchronous scheduler, $\mathcal{S}_{\texttt{Prob}}$ is a* probabilistic self-stabilizing *system for $\mathcal{SP}$.*

Using the same approach as for Theorem 8, the following result is straighforward.

Theorem 9 *Assuming a distributed randomized scheduler, $\mathcal{S}_{\texttt{Prob}}$ is a* probabilistic self-stabilizing *system for $\mathcal{SP}$.*

5 Conclusion

Weak-stabilization is a variant of self-stabilization that only requires the *possibility* of convergence, thus enabling to solve problems that are otherwise impossible to solve with self-stabilizing guarantees. As seen throughout the paper, weak-stabilizing protocols are much easier to design and prove than their self-stabilizing counterparts. Yet, the main result of the paper is the practical impact of weak-stabilization: all deterministic weak-stabilizing algorithms can automatically be turned into probabilistic self-stabilizing ones, provided the scheduling is probabilistic (which is indeed the case for practical purposes). Our approach removes the burden of designing and proving probabilistic stabilization by algorithms designers, leaving them with the easier task of designing weak stabilizing algorithms.

Although this paper mainly focused on the theoretical power of weak-stabilization, a goal for future research is the quantitative study of weak-stabilization, evaluating the expected stabilization time of transformed algorithms.

References

[1] D. Angluin. Local and global properties in networks of processes. In *12th Annual ACM Symposium on Theory of Computing*, pages 82–93, April 1980.

[2] Joffroy Beauquier, Christophe Genolini, and Shay Kutten. -stabilization of reactive tasks. In *PODC*, page 318, 1998.

[3] 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.

[4] Steven C. Bruell, Sukumar Ghosh, Mehmet Hakan Karaata, and Sriram V. Pemmaraju. Self-stabilizing algorithms for finding centers and medians of trees. *SIAM J. Comput.*, 29(2):600–614, 1999.

[5] F. Buckley and F Harary. *Distance in Graphs*. Addison-Wesley Publishing Compagny, Redwood City, CA, 1990.

[6] J. Burns, M. Gouda, and R. Miller. On relaxing interleaving assumptions. *Proceedings of the MCC Workshop on Self-Stabilizing Systems, Austin, Texas*, 1989.

[7] James E. Burns, Mohamed G. Gouda, and Raymond E. Miller. Stabilization and pseudo-stabilization. *Distrib. Comput.*, 7(1):35–42, 1993.

[8] Anurag Dasgupta, Sukumar Ghosh, and Xin Xiao. Probabilistic fault-containment. In *Stabilization, Safety, and Security of Distributed Systems, 9th International Symposium, SSS*, volume 4838 of *Lecture Notes in Computer Science*, pages 189–203. Springer, 2007.

[9] Ajoy K. Datta, Maria Gradinariu, and Sébastien Tixeuil. Self-stabilizing mutual exclusion with arbitrary scheduler. *The Computer Journal*, 47(3):289–298, 2004.

[10] EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.

[11] Shlomi Dolev. *Self-Stabilization*. The MIT Press, March 2000.

[12] Christophe Genolini and Sébastien Tixeuil. A lower bound on k-stabilization in asynchronous systems. In *Proceedings of IEEE 21st Symposium on Reliable Distributed Systems (SRDS'2002)*, Osaka, Japan, October 2002.

[13] Mohamed G. Gouda. The theory of weak stabilization. In *WSS*, pages 114–123, 2001.

[14] Maria Gradinariu and Sébastien Tixeuil. Conflict managers for self-stabilization without fairness assumption. In *27th IEEE International Conference on Distributed Computing Systems (ICDCS)*, page 46. IEEE Computer Society, 2007.

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

[16] Ted Herman. Probabilistic self-stabilization. *Inf. Process. Lett.*, 35(2):63–67, 1990.

[17] Amos Israeli and Marc Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *PODC*, pages 119–131, 1990.

[18] Sébastien Tixeuil. *Wireless Ad Hoc and Sensor Networks*, chapter Fault-tolerant distributed algorithms for scalable systems. ISTE, October 2007. ISBN: 978 1 905209 86.

A Proof of Theorem 2

Definition 8 (TokenHolders) *Let γ be a configuration. Let* $\texttt{TokenHolders}(\gamma)$ *be the set of processes p satisfying $Token(p)$ in the configuration γ.*

Definition 9 ($\mathcal{LCSET}$) *Let $\mathcal{LCSET}$ be the set of configurations γ such that γ satisfies* $|\texttt{TokenHolders}(\gamma)| = 1$.

Definition 10 (PredPath) *Let p and q be two distinct processes. We call* $\texttt{PredPath}(p, q)$ *be the unique path $p_0, \ldots, p_k$ such that: (1) $p_0 = p$, (2) $\forall i \in [1 \ldots k]$, $Pred_{p_k} = p_{k-1}$, and $p_k = q$.*

Remark 1 *Let p and q be two distinct processes.* $\texttt{PredPath}(p, q) \neq \texttt{PredPath}(p, q)$.

Definition 11 (MTD: MinTokenDistance) *Let γ be a configuration such that γ satisfies* $|\texttt{TokenHolders}(\gamma)| > 1$. *We denote by* $\texttt{MTD}(\gamma)$ *the length of the shortest path* $\texttt{PredPath}(p, q)$ *such that $Token(p)$ and $Token(q)$ in γ.*

Lemma 4 *For any configuration γ, we have* $|\texttt{TokenHolders}(\gamma)| > 0$.

Proof. Assume, by the contradiction, that there is a configuration γ such that $|\texttt{TokenHolders}(\gamma)| = 0$. Let $p_0, \ldots, p_{N-1}$ be an hamiltonian path of processes such that, $\forall i \in [0 \ldots N-1]$, $p_i = Pred_{p_{(i+1) \bmod N}}$. Then, $(\forall i \in [0 \ldots N-1], \neg Token(p_i))$ implies that $(\forall i \in [0 \ldots N-1], [dt_{p_{(i+1) \bmod N}} = ((dt_{p_i} + 1) \bmod m_N)])$ which is not possible because $(N \bmod m_N) \neq 0$, a contradiction. □

Lemma 5 (Possible Convergence) *Starting from any configuration, there exists at least one possible execution that reaches a configuration $\gamma \in \mathcal{LCSET}$.*

Proof. Any configuration satisfies $|\texttt{TokenHolders}| > 0$ by Lemma 4. Consider any configuration γ satisfying $|\texttt{TokenHolders}(\gamma)| > 1$. Let us study the two following cases:

$\texttt{MTD}(\gamma) = 1$. In this case, there exists two processes p and q such that $|\texttt{PredPath}(p,q)| = 1$, *i.e.*, p is the predecessor of q and both p and q satisfies the predicate $Token$ (*i.e.*, both p and q hold a token). If only p executes Action A in the next step, then p satisfies $\neg Token$ in the next configuration γ' and, as the consequence, $|\texttt{TokenHolders}(\gamma')| < |\texttt{TokenHolders}(\gamma)|$.

$\texttt{MTD}(\gamma) > 1$. Let consider two processes p and q such that $|\texttt{PredPath}(p,q)| = \texttt{MTD}(\gamma)$. Then, Action A is enabled at p and if only p moves in the next step, then $|\texttt{PredPath}(p,q)|$

decreases of one unit in the next configuration. Hence, inductively there exists an execution from γ that reaches a configuration γ' such that $\texttt{MTD}(\gamma') = 1$.

Hence, from any configuration γ such that $|\texttt{TokenHolders}(\gamma)| > 1$ there always exists an execution where the cardinal of `TokenHolders` eventually decreases and the lemma is proven. □

Lemma 6 (Strong Closure) *Any execution starting from a configuration γ such that $\gamma \in \mathcal{LCSET}$ always satisfies the specification of the token circulation.*

Proof. To prove this lemma we show that $\forall \gamma \in \mathcal{LCSET}$, $\forall \gamma \mapsto \gamma'$, (1) $\gamma' \in \mathcal{LCSET}$ ($\mathcal{LCSET}$ is closed) and (2) the token holder in γ' is the successor of the token holder in γ.

Consider a configuration γ such that $|\texttt{TokenHolders}(\gamma)| = 1$. Let q be the only process satisfying $Token(q)$ in γ. Let p and s be the predecessor and the successor of q in γ, respectively. Then, in γ, q is the only enabled process, $dt_q \neq ((dt_p + 1) \bmod m_N)$, and $dt_s = ((dt_q + 1) \bmod m_N)$. During the next step, q executes `A` and, as a consequence, $dt_q = ((dt_p + 1) \bmod m_N)$ and $dt_s \neq ((dt_q + 1) \bmod m_N)$ in the next configuration γ': s is the only token holder in γ', which proves the lemma. □

Proof of Theorem 2. By Lemmas 5 and 6, the theorem is obvious. □

B Proof of Theorem 4

Definition 12 (`ParPath`) *We call* $\texttt{ParPath}(p)$ *the* unique maximal *path $p_0, \ldots, p_k$ such that: (1) $p_k = p$, (2) $\forall i \in [1 \ldots k]$, $Par_i = p_{i-1}$, and (3) p_0 satisfies $[(Par_{p_0} \neq \perp) \Rightarrow (Par_{Par_{p_0}} = p_0)]$.*

Notation 1 *Let p be a process. In the following, we denote by* $\texttt{Root}(p)$ *the* initial extremity *of* $\texttt{ParPath}(p)$ *(*n.b.*, $(Par_p = \perp) \Rightarrow (\texttt{Root}(p) = p)$).*

Remark 2 *As the network is acyclic, for any process p,* $\texttt{ParPath}(p)$ *has a finite length.*

Definition 13 ($\mathcal{LC}$) *Any configuration γ satisfies the predicate $\mathcal{LC}(\gamma)$* iff *the two following conditions hold in γ: (1) there exists exactly one process p that satisfies $Par_p = \perp$ and (2) for any process $q \neq p$, $\texttt{Root}(q) = p$.*

Remark 3 *There is exactly one process satisfying $isLeader$ in any configuration γ satisfying $\mathcal{LC}(\gamma)$.*

Lemma 7 *In any configuration where every process satisfies $\neg isLeader$, there exists at least one process p such that Action* $\texttt{A}_1$ *is enabled at p.*

Proof. Let $NearestCenter(p)$ be the center process at the smallest distance from the process p. Let $\mathcal{DNC}_{max} = \lceil D/2 \rceil$ be the maximal distance between any process p and $NearestCenter(p)$. Let $\mathcal{DNC}^{-1}(p) = \mathcal{DNC}_{max} - d(p, NearestCenter(p))$.

Assume, by the contradiction, that there exists a configuration γ where every process satisfies $\neg isLeader$ and no Action $\mathtt{A_1}$ is enabled. We show the contradiction in two steps:

Step 1. First, we prove that any process p such that $\mathcal{DNC}^{-1}(p) = d$ with $0 \leq d < \mathcal{DNC}_{max}$ (actually the non-center processes) satisfies $Par_p = q$ in γ with $\mathcal{DNC}^{-1}(q) = d+1$.

Step 2. Then, we show the contradiction using **Step 1**.

Step 1. (by induction)

Induction for $d = 0$. By Definition, any process p such that $\mathcal{DNC}^{-1}(p) = 0$ is a leaf node. As p satisfies $\neg isLeader(p)$, $Par_p = q$ holds in γ where q is the only neighbor of p. Now, by definition, $\mathcal{DNC}^{-1}(q) = \mathcal{DNC}^{-1}(p) + 1 = 1$. Hence, the induction holds for $d = 0$.

Induction Assumption: Let $k \in [0 \dots \mathcal{DNC}_{max} - 1]$. Assume that any process p such that $0 \leq \mathcal{DNC}^{-1}(p) < k$ satisfies $Par_p = q$ in γ with $\mathcal{DNC}^{-1}(q) = k + 1$.

Induction for $d = k + 1$. Consider a process p such that $\mathcal{DNC}^{-1}(p) = k + 1$. Then, $\mathcal{DNC}^{-1}(p) < \mathcal{DNC}_{max}$ and, by definition, p has one neighbor q such that $\mathcal{DNC}^{-1}(q) = k+2$ and all its other neighbors q' satisfies $\mathcal{DNC}^{-1}(q') = k$. Assume, by the contradiction, that $Par_q = v$ with $\mathcal{DNC}^{-1}(v) = k$. Then, any other v's neighbor, v', satisfies $\mathcal{DNC}^{-1}(v) = k-1$. Hence, by induction assumption, any process v' satisfies $Par_{v'} = v$. Now, $Par_v \neq \perp$ because v satisfies $\neg isLeader(v)$. So, Action $\mathtt{A_1}$ is enabled at v, a contradiction. Hence, $Par_p = q$ where q is the only neighbor of p such that $\mathcal{DNC}^{-1}(q) = k + 2$ and the induction holds for $d = k + 1$.

Step 2.

We now show the contradiction. By Property 1 (page 4), we can split our study in the two following cases:

There is one center c in the network. In this case, any neighbor of c, c', satisfies $\mathcal{DNC}^{-1}(c') = \mathcal{DNC}_{max} - 1$. In this case, any process c' also satisfies $Par_{c'} = c$ (**Step 1**). Now, $Par_c \neq \perp$ because c satisfies $\neg isLeader(c)$. So, Action $\mathtt{A_1}$ is enabled at c, a contradiction.

There is two neighboring centers c_0 and c_1 in the network. In this case, any non-center neighbor of c_i ($i \in \{0, 1\}$), c'_i, satisfies $\mathcal{DNC}^{-1}(c'_i) = \mathcal{DNC}_{max} - 1$. In this case, any process c'_i also satisfies $Par_{c'_i} = c_i$ (**Step 1**). Assume now, by the contradiction, that one the centers c_i ($i \in \{0, 1\}$) satisfies $Par_{c_i} = c'_i$ where c'_i is a neighbor such that $\mathcal{DNC}^{-1}(c'_i) = \mathcal{DNC}_{max} - 1$. Then, any other c'_i's neighbor also satisfies $Par = c'_i$ (**Step 1**). Now, $Par_{c'_i} \neq \perp$ because c'_i satisfies $\neg isLeader(c'_i)$. So, Action $\mathtt{A_1}$ is enabled at c'_i, a contradiction. Hence, $Par_{c_0} = c_1$ and $Par_{c_1} = c_0$ and Action $\mathtt{A_1}$ is both enabled at c_0 and c_1, a contradiction. □

The following corollary simply holds by the fact that after executing Action $\mathtt{A_1}$, a process satisfies $isLeader$.

Corollary 1 *Starting from any configuration, the system can reach in at most one step a configuration where at least one process satisfies isLeader.*

Lemma 8 *From any configuration where at least one process satisfies isLeader, there is a possible execution that reaches a configuration γ satisfying $\mathcal{LC}(\gamma)$.*

Proof. Let p be a process satisfying $isLeader(p)$. Let $Tree(p) = \{q \in V, \texttt{Root}(q) = p\}$. First, from Definition 13, we can trivially deduce that a configuration satisfies $\mathcal{LC}$ *iff* it contains a unique tree $Tree(p)$ such that $Tree(p) = V$.

Consider then a configuration γ satisfying $\neg\mathcal{LC}(\gamma)$ where there exists a process p satisfying $isLeader(p)$. So, $Tree(p) \subset V$. Let $NonTree(p) = V \setminus Tree(p)$. To prove this lemma, we just show below that from such a configuration γ is always possible to reach (in a finite number of step) a configuration γ' where the cardinal of $NonTree(p)$ decreased.

First, p satisfying $isLeader(p)$ in γ, so, $Tree(p) \neq \emptyset$ in γ. Then, as γ satisfies $\neg\mathcal{LC}(\gamma)$, $NonTree(p) \neq \emptyset$ and, as the network is connected, there two neighboring processes v and w such that $v \in Tree(p)$ and $w \in NonTree(p)$ in γ. Also, $Par_v \neq w$ and $Par_w \neq v$ in γ by Definition 12. Consider then the two following cases:

- $Par_w \neq \perp$ *in* γ. In this case, Action $\texttt{A}_2$ is enabled at w until (at least) $Par_w = v$. Now, after at most $\Delta_w - 1$ executions of Action $\texttt{A}_2$, Par_w points out to v. Hence, if only actions $\texttt{A}_2$ at w are executed until Par_w points out to v, there is an execution from γ that reaches a configuration γ' where $|NonTree(p)|$ decreases of one unit.

- $Par_w = \perp$ *in* γ. In this case, as $Par_v \neq w$, Action $\texttt{A}_3$ is enabled at w. If only w moves in the next step, then either (1) Par_w points out to v in the next configuration and $|NonTree(p)|$ decreases of one unit, or (2) $Par_w \notin \{v,\perp\}$ in the next configuration and we retreive the previous case.

Hence, from any configuration γ satisfying $\neg\mathcal{LC}(\gamma)$ where there is a process p satisfying $isLeader(p)$, it is always possible to reach a configuration γ' where $|NonTree(p)|$ decreased. □

By Corollary 1 and Lemma 8, follows:

Lemma 9 (Possible Convergence) *Starting from any configuration, there exists at least one possible execution that reaches a configuration γ satisfying $\mathcal{LC}(\gamma)$.*

Lemma 10 (Strong Closure) *Let γ be a configuration. γ satisfies $\mathcal{LC}(\gamma)$* iff *γ is a* terminal configuration*.*

Proof.

If. Consider a configuration γ satisfying $\mathcal{LC}(\gamma)$. Let p be the only process that satisfies $Par = \perp$ in γ. By Definition 13, any neighbor p' of p satisfies $Par_{p'} = p$ and, as a consequence, p is disabled. Consider now any process q such that $Par_q \neq \perp$. As we are in a tree network, there is only one path linking any process q to p, so, by Definition 13, any process q points out with Par_q the unique neighbor q' whereby it can reach p, q' does not point out to q with $Par_{q'}$, and all other neighbors of q points out to q with their Par pointer. As a consequence, any process q is disabled. Hence, γ is a terminal configuration.

Only If. (by the contraposition) By Lemma 9, any configuration γ satisfying $\neg\mathcal{LC}(\gamma)$ is not *terminal*. □

Proof of Theorem 4. Follows from lemmas 9 and 10, and Remark 3. □

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399