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

Stéphane Devismes
CNRS
Université Paris-Sud

Sébastien Tixeuil
Université Paris 6
LIP6-CNRS & INRIA

Masafumi Yamashita
CSCE
Kyushu University

## Abstract

*Self-stabilization is a strong property which guarantees that a network always resume a 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 algorithms are easier to design and prove than their (probabilistic) self-stabilizing counterparts.*

## 1. Introduction

Self-stabilization [11, 12] 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* [18] 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, 13] 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* [14] 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 [17] or to solve problems that are known to be impossible to solve in the classical deterministic setting [15], 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, [19] 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 [14] 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.

Due to the lack of space, many technical proofs have been removed from the paper. These proofs are available in the technical report [10].

## 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* $\{p,q\} \in E$. $\Gamma_p$ denotes the set of $p$'s neighbors. $\Delta_p$ denotes the *degree* of $p$, *i.e.*, $|\Gamma_p|$. By extention, we denote by $\Delta$ 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 indices*, 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 $\gamma$ *iff* its guard is $true$. By extention, $p$ is said *enabled* in $\gamma$ *iff* at least one of its action is enabled in $\gamma$.

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 $\gamma$ is said *terminal* if there is no configuration $\gamma'$ such that $\gamma \mapsto \gamma'$. We denote by $\gamma \rightsquigarrow \gamma'$ the fact that $\gamma'$ is *reachable* from $\gamma$, *i.e.*, there exists an execution starting from $\gamma$ and containing $\gamma'$.

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* [11] 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 ([12]). 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 $\gamma$ 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** *$\mathcal{S}$ 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** *$\mathcal{S}$ 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** *$\mathcal{S}$ 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 $\gamma$, there exists an execution starting from $\gamma$ 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 claimed below.

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

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** *The* token circulation *problem consists in circulating a single token 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). 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]).

**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 :: $Token(p) \quad \rightarrow \quad PassToken_p$

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 *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 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.*

## 3.2. Leader Election

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

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

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

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 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:

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 $A_1$), *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

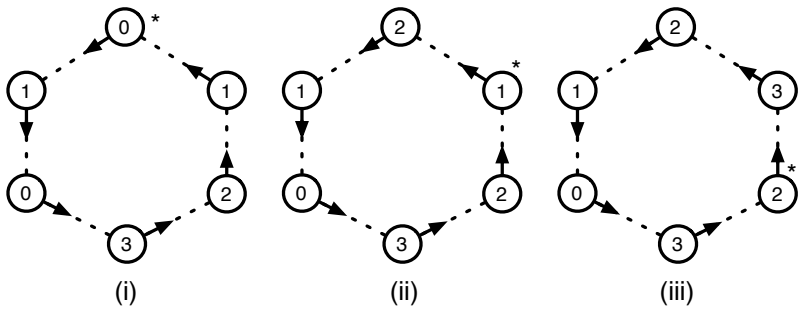


**Figure 1. Example of an execution starting from a legitimate configuration.**

**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:**

| | | | | |
|---|---|---|---|---|
| $\mathtt{A}_1$ | :: | $(Par_p \neq \perp) \wedge (\lvert Children_p \rvert = \lvert Neig_p \rvert)$ | $\rightarrow$ | $Par_p \leftarrow \perp$ |
| $\mathtt{A}_2$ | :: | $(Par_p \neq \perp) \wedge [Neig_p \setminus (Children_p \cup \{Par_p\}) \neq \emptyset]$ | $\rightarrow$ | $Par_p \leftarrow (Par_p + 1) \bmod \Delta_p$ |
| $\mathtt{A}_3$ | :: | $(Par_p = \perp) \wedge (\lvert Children_p \rvert < \lvert Neig_p \rvert)$ | $\rightarrow$ | $Par_p \leftarrow \min_{\prec_p}(Neig_p \setminus Children_p)$ |

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 $\Delta_p$ (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 $\mathtt{A}_\mathtt{j}$ 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 $(v)$.

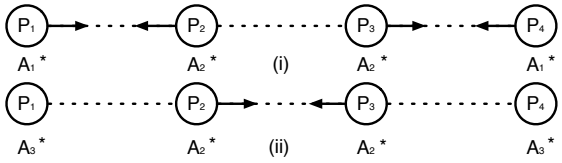


**Figure 3. Example of an execution that does not converge.**

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 $(i)$,

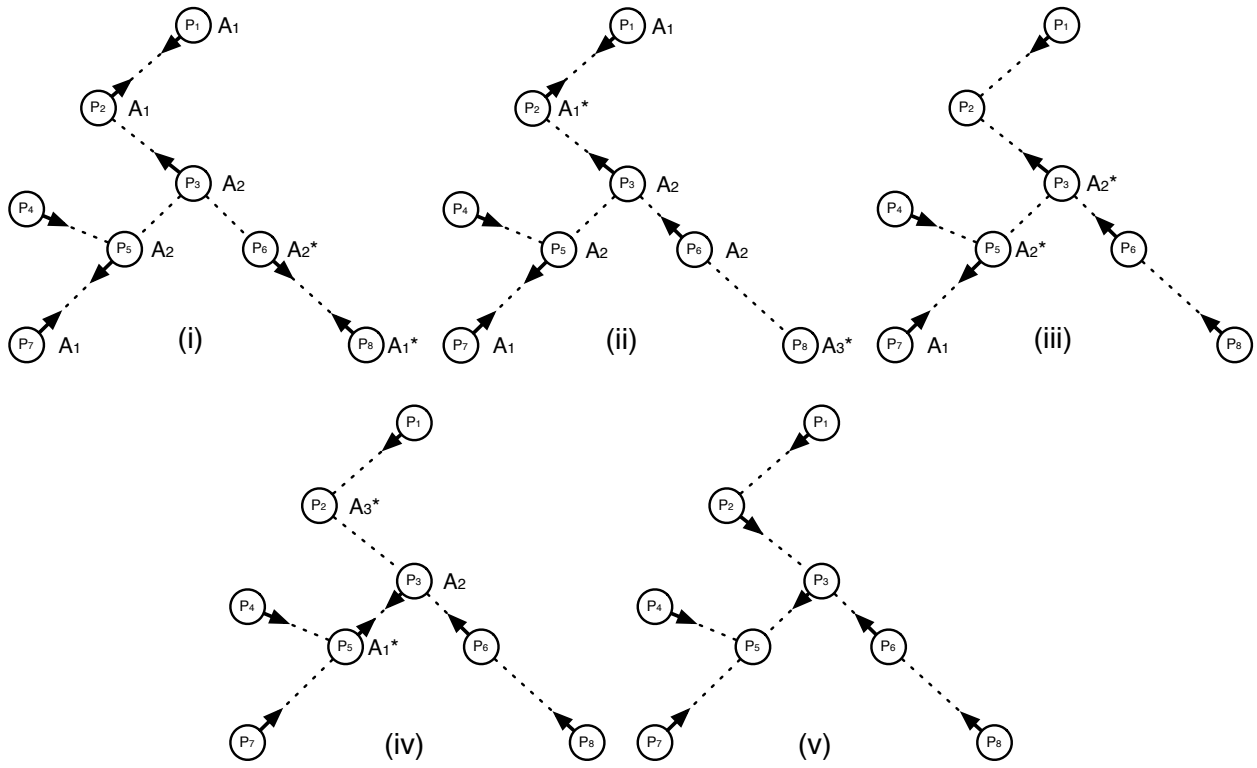


**Figure 2. Example of possible convergence.**

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.*

## 4. From Weak to Probabilistic Stabilization

In [14], Gouda shows that deterministic weak-stabilization is a "good approximation" of deterministic self-stabilization[1] by proving the following theorem:

**Theorem 5 ([14])** *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 $\gamma$ 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 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 4) is a deterministic weak-stabilizing token circulation with a finite number of con-

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

figurations, 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 $\gamma_0$ where only $p_0$ and $p_3$ hold a token. Both $p_0$ and $p_3$ are enabled in $\gamma_0$. Assume that only $p_0$ passes its token in the step $\gamma_0 \mapsto \gamma_1$. In $\gamma_1$, $p_1$ 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* is a notion that is, in some sense, equivalent to the *Gouda's strong fairness*.

**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 $\gamma_0$ 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 $\gamma_k$ 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 $\gamma_0$. Hence, $\gamma_0 \mapsto \gamma_1$ occurs with a probability 1 after a finite number of occurences of $\gamma_0$ in $e$ and, as a consequence, $\gamma_1$ occurs infinitely often (with the probability 1) in $e$. Inductively, it is then straightforward that $\forall i \in [1 \ldots k]$, $\gamma_i$ occurs infinitely often in $e$ with the probability 1. Hence, the legitimate configuration $\gamma_k$ 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 5) 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 symmetries 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 Symetries.** 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 [15]): 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 $\mathtt{A} :: Guard_{\mathtt{A}} \rightarrow \mathtt{S}_{\mathtt{A}}$ of the input (deterministic weak-stabilizing) algorithm into the following action $\mathtt{Trans(A)}$:

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

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.

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 $\mathcal{S}_{\mathtt{Prob}} = (\mathcal{C}_{\mathtt{Prob}},\mapsto_{\mathtt{Prob}})$ be the probabilistic system obtained by transforming $\mathcal{S}_{\mathtt{Det}}$ according to the above presented method. We have the two following theorems:

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

**Theorem 9** *Assuming a distributed randomized scheduler, $\mathcal{S}_{\mathtt{Prob}}$ 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] J. Beauquier, C. Genolini, and S. Kutten. $k$-stabilization of reactive tasks. In *PODC*, page 318, 1998.

[3] J. Beauquier, M. Gradinariu, and C. Johnen. Randomized self-stabilizing and space optimal leader election under arbitrary scheduler on rings. *Dist. Comp.*, 20(1):75–93, 2007.

[4] S. C. Bruell, S. Ghosh, M. H. Karaata, and S. 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 WSS, Austin, Texas*, 1989.

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

[8] A. Dasgupta, S. Ghosh, and X. Xiao. Probabilistic fault-containment. In *SSS'07*, volume 4838 of *LNCS*, pages 189–203. Springer, 2007.

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

[10] S. Devismes, S. Tixeuil, and M. Yamashita. Weak vs. self vs. probabilistic stabilization. *CoRR*, abs/0711.3672, 2007.

[11] E. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

[12] S. Dolev. *Self-Stabilization*. The MIT Press, March 2000.

[13] C. Genolini and S. Tixeuil. A lower bound on $k$-stabilization in asynchronous systems. In *Proceedings of SRDS'2002*, Osaka, Japan, October 2002.

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

[15] M. Gradinariu and S. Tixeuil. Conflict managers for self-stabilization without fairness assumption. In *ICDCS'07*, page 46. IEEE Computer Society, 2007.

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

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

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

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