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

Stéphane Devismes Sébastien Tixeuil Masafumi Yamashita

January 24, 2021

Abstract

Self-stabilization is a strong property, which guarantees that a distributed system always resumes a correct behavior starting from an arbitrary initial state. Since it is a strong property, some problems cannot have self-stabilizing solutions. Weaker guarantees hence have been later introduced to cope with impossibility results, *e.g.*, probabilistic self-stabilization only guarantees probabilistic convergence to a correct behavior, and weak stabilization only guarantees the possibility of convergence. In this paper, we investigate the relative power of self, probabilistic, and weak stabilization, with respect to the set of problems that can be solved. Weak stabilization is by definition stronger than self-stabilization and probabilistic self-stabilization in that precise sense. We first show that weak stabilization allows to solve problems having no self-stabilizing solution. We then show that any *finite state* deterministic weak stabilizing algorithm to solve a problem under the strongly fair scheduler is always a probabilistic self-stabilizing algorithm to solve the same problem under the randomized scheduler. Unfortunately, this good property does not hold in general for *infinite state* algorithms. We however show that for some classes of infinite state algorithms, this property holds. These results hint at more practical use of weak stabilizing algorithms, as they are easier to design and prove their correctness than their self-stabilizing and probabilistic self-stabilizing counterparts.

keywords: Distributed systems; distributed algorithm; fault-tolerance; self-stabilization; weak stabilization; probabilistic self-stabilization.

1 Introduction

Self-stabilization [2–4] is a versatile technique to withstand *any* finite number of transient faults in a distributed system. Informally, an algorithm is self-stabilizing if, starting from *any* initial configuration, *every* execution eventually reaches a point from which its behavior is correct. The arbitrary initial configuration models any configuration which can be reached by the system immediately after the occurrence of transient faults. Thus, self-stabilization makes no hypothesis 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 algorithms may require a large amount of resources, may be difficult to design and prove, and are unable to solve some fundamental problems in distributed computing. To cope with those issues, several weakened forms of self-stabilization have been proposed in the literature.

Probabilistic self-stabilization [5] 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. Some problems such as graph coloring and token passing on anonymous networks are known to be impossible to solve under the deterministic self-stabilizing setting, but there are probabilistic self-stabilizing algorithms for them [5–7]. Probabilistic self-stabilization is also used to reduce resource consumption [8].

Pseudo-stabilization [9] 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 may

*A preliminary version of this work was published in [1].

be unbounded. Burns *et al* [9] shows that the alternating bit protocol is an example of pseudo-stabilizing algorithm that is not self-stabilizing.

The notion of *k-stabilization* [4, 10–12] prohibits some of the configurations from being possible initial states, as an initial configuration may only be the result of at most k faults, where the number of faults is defined as the number of process memories to change to reach a correct configuration. For the token circulation problem, two k-stabilizing algorithms are given that guarantee a small convergence time depending only on k (not the size of system) [10]. In [12], authors propose a k-stabilizing solution for the leader recovery problem in anonymous networks. This problem has no (deterministic) self-stabilizing solution.

Finally, *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. It is only known that a sufficient condition on the scheduling hypothesis makes a weak stabilizing solution self-stabilizing [13].

From a problem-centric point of view, probabilistic, pseudo, and k variants of self-stabilization have been demonstrated to be strictly more powerful than classical self-stabilization, which comforts the intuition that they provide weaker guarantees. In contrast, no such knowledge is available regarding weak stabilization.

In this paper, our goal is not to propose particular instance of weak stabilizing algorithms. Merely, we want to show some general properties about weak-stabilization. In particular, we address the open question of the power of weak stabilization with respect to the set of problems that can be solved. Our contribution is three-folds:

1. We first prove that from the problem-centric point of view, weak stabilization is strictly stronger than self-stabilization, both for static problems, such as leader election, and for dynamic problems, such as token passing.

2. We then show that there is a relationship between deterministically weak stabilizing algorithms and probabilistically self-stabilizing ones: Any finite state deterministically weak stabilizing algorithm to solve a problem under the strongly fair scheduler is always a probabilistically self-stabilizing algorithm to solve the same problem under the randomized scheduler.[1] Unfortunately, infinite state algorithms do not have this good property. We however show this property for several natural classes of algorithms.

3. We also propose a transformer that transforms any finite state deterministically weak stabilizing algorithm to solve a problem under the strongly fair scheduler into a randomized algorithm that is probabilistically self-stabilizing to solve the same problem under the synchronous scheduler.[2]

These results have practical impact, since it is much easier to design a weak stabilizing algorithm and to prove its correctness than a probabilistic self-stabilizing algorithm (as we shall see in Section 3), and our scheme automatically makes them self-stabilizing in the probabilistic sense when new simple weak stabilizing solutions appear.

Paper Outline. The remainder of the paper is organized as follows. After providing the notions and notations in the next section, we present in Section 3 very simple weak stabilizing algorithms for the token circulation and the leader election problems on anonymous systems, for which there are no deterministic self-stabilizing algorithms. In Section 4, we then show that some (finite and infinite state) weak stabilizing algorithms can be transformed into probabilistic self-stabilizing algorithms. In Section 5, we discuss about time complexity issues. We make some concluding remarks in Section 6.

2 Preliminaries

2.1 Graphs

A *digraph* G is a pair (V,E), where V is a set of *nodes* and E is a set of directed *edges*. Each edge is an ordered pair of two nodes. Self-loops are allowed to exist. A node p is a *predecessor* of a node q if $(p,q) \in E$.

[1] For the definitions of the strongly fair and the randomized scheduler, see Section 2 and Subsection 4.1.

[2] For the definition of the synchronous scheduler, see Section 2.

Let $\Gamma_q^- = \{p : (p,q) \in E\}$ be the set of predecessors of q. The number $\delta_q^- = |\Gamma_q^-|$ of the predecessors of q is called the *indegree* of q. Let $\Gamma_q^+ = \{p : (q,p) \in E\}$ be the set of *successors* of q. $\delta_q^+ = |\Gamma_q^+|$ is called the *outdegree* of q.

A *(directed) path* of length k is a sequence $P = p_0, p_1, \ldots, p_k$ of $k+1$ nodes in V such that $(p_i, p_{i+1}) \in E$ for all $0 \leq i \leq k-1$. The *distance* $d(p,q)$ between two nodes p and q is the length of a shortest path between p and q in G. G is said to be *strongly connected* if there is a path between every pair of nodes. A path $P = p_0, p_1, \ldots, p_k$ is said to be *simple* if $p_i \neq p_j$ holds for all $0 \leq i < j \leq k$. A path $P = p_0, p_1, \ldots, p_k$ is called a *directed (simple) cycle* if path $P = p_0, p_1, \ldots, p_{k-1}$ is simple and $p_0 = p_k$. A finite digraph consisting only of a single directed cycle is called a *(unidirectional) ring.*

A digraph $G = (V, E)$ is said to be *undirected* if $(p,q) \in E$ implies $(q,p) \in E$ for all $p, q \in V$. Note that every pair of directed edges (p,q) and (q,p) are identified as a single undirected edge in an undirected graph. In an undirected graph, a path $P = p_0, p_1, \ldots, p_k$ is said to be a *(simple) cycle* if $p_0 = p_k$ and for all $0 \leq i < j < k$, $p_i \neq p_j$ and $(p_i, p_{i+1}) \neq (p_j, p_{j+1})$. A node p is called a *neighbor* of a node q, if $(p,q) \in E$. Let Γ_q and δ_q be respectively the set $\{p : (p,q) \in E\}$ of the neighbors of $q \in V$ and the degree $|\Gamma_q|$ of q. We mean an undirected graph by a graph.

A graph G is *connected* if there is a path in G between every pair of nodes. A connected acyclic finite graph is called a *(finite) tree.* In a tree, we distinguish two types of nodes; the *leaves* (*i.e.*, any node p such that $\delta_p = 1$) and the *internal nodes* (*i.e.*, any node p such that $\delta_p > 1$). The eccentricity $ec(p)$ of a node p is $\max_{q \in V} d(p,q)$. A node p is a *center* of G if $ec(p) \leq ec(q)$ for all $q \in V$. The next property is well-known [14].

Property 1 *A tree has a unique center or two neighboring centers.*

2.2 Anonymous Distributed Systems

Consider a distributed system consisting of N communicating processes. As usual, we model its communication network by a digraph $G = (V,E)$, where V and E respectively represent the sets of the processes and the unidirectional communication links, *i.e.*, an edge (p,q) is in E if and only if process q can directly obtain information from process p. Note that p can of course directly obtain information from p itself, but we do not include self-loop (p,p) in E for simplicity: G does not contain self-loops.

In this paper, we propose algorithms for *anonymous* distributed systems. The processes do not have unique names (identifiers) and may only be differentiated by their indegrees and/or outdegrees. Process names $p \in V$ are used only for the purpose of explanation. A process $q \in V$ (and algorithm) thus identifies each of its predecessors in Γ_q^- using *local indices* $L_q = \{0, 1, \ldots, \delta_q^- - 1\}$.

Communication is carried out by means of *shared variables.* Each process p holds a set of shared variables and can access them to read and to write. The successors of p can access them only to read. Suppose that a variable v is provided in each process in V. We denote the variable v at a process p by v_p. If v_p is a shared variable and $(p,q) \in E$, then q can read the value of v_p. However q cannot use p to specify process p. It hence specifies v_p as v_{i_p}, *i.e.*, as the variable v of the predecessor whose index is $i_p \in L_q$.

A variable in a process takes a value from a pre-determined domain. The state of a process is a function (assignment) that assigns, to each of the variables in the process, a value in its domain. Let S_p be the set of all states of a process $p \in V$. Then the Cartesian product $\Pi_{p \in V} S_p$ of S_p for all $p \in V$ forms the set of all *(global) configurations*; a configuration describes the values of all variables in the system. The above definition of configuration implicitly assumes that read and write operations are *atomic* and finish at a time even when a read operation is initiated by a neighbor, unlike the message passing model, in which we also need the state of each communication link to describe the configuration of system.

A process p can change its state by executing its *(local) algorithm* specified by a sequence of guarded actions of the form:

$$\langle label \rangle :: \langle guard \rangle \rightarrow \langle statement \rangle.$$

The guard of an action at p is a boolean expression that involves some variables of p and its predecessors. An action can be executed only if its guard is true, and the statement of the action updates some variables of p. We assume that the execution of any action is atomic.

An action of some process p is said to be enabled in a configuration γ if its guard is true. Without loss of generality, we assume that at most one action is enabled at p in γ. We say that p is *enabled* in γ if one of its actions is enabled in γ. Simultaneous executions of actions by a set of processes induce a transition between two configurations. If a configuration γ' yields from a configuration γ, we denote this transition by $\gamma \mapsto \gamma'$. For the consistency, we allow $\gamma \mapsto \gamma$ for any configuration γ.

Given a communication network G and an algorithm $\mathcal{A}$, the set $\mathcal{C}$ of configurations and the transition relation $\mapsto$ are determined. We model the (behavior of) distributed system executing $\mathcal{A}$ on G by a state transition system (*i.e.*, a digraph) $\mathcal{S} = (\mathcal{C},\mapsto)$. An *execution* of $\mathcal{S}$ is an infinite (directed) path $\gamma_0, \gamma_1, \ldots$ starting from a configuration γ_0 in $\mathcal{C}$. We call each of $\gamma_{i-1} \mapsto \gamma_i$ a *step*. A configuration γ is said to be *terminal* if no processes are enabled in γ. Remark that the only possible execution starting from a terminal configuration γ is γ^ω, *i.e.*, an infinite sequence $\gamma, \gamma, \ldots$ We say that a configuration γ' is *reachable* from a configuration γ, if there is a directed path from γ to γ' in $\mathcal{S}$.

In general, there is more than one execution $\xi = \gamma_0, \gamma_1, \ldots$ in $\mathcal{S}$ for some initial configuration $\gamma_0 \in \mathcal{C}$. A *scheduler* determines which executions among them are executable by non-deterministically choosing some processes to have them execute the corresponding actions from the processes enabled in γ_t, at each time instant t. We define a scheduler as a predicate σ over the executions. We denote by $\mathcal{E}(\mathcal{S}, \sigma)$ the set of all executions that can occur as an execution of $\mathcal{S}$ under σ.

The scheduler that always activates all enabled processes is called the *synchronous* scheduler. A scheduler that always activates exactly one process is called a *central* scheduler. Every scheduler that is not central is said to be *distributed*. In this paper, unless otherwise specified, the scheduler is always assumed to be distributed.

Then, a scheduler may have some *fairness* properties [15]. A scheduler is *proper*, if it always chooses at least one process as long as there are enabled processes. A scheduler is *weakly fair* if every *continuously* enabled process is eventually chosen to execute an action. A scheduler is *strongly fair* if every process that is enabled *infinitely often* is eventually chosen to execute an action. Notice that we will also use a stronger version of strongly fairness introduced by Gouda in [13], hence called *Gouda's strong fairness*: A scheduler is *Gouda's strongly fair* if for every transition $\gamma \mapsto \gamma'$, if γ occurs infinitely often in an execution ξ, then $\gamma \mapsto \gamma'$ also appears infinitely often in ξ. Finally, the *randomized scheduler* always chooses each of the enabled processes independently at random with probability 1/2. Hence the randomized scheduler is not proper, since it may not activate any process even if there are enabled ones.

Note that predicate f defined over a set D is said to be larger than another predicate g, if $g(d)$ implies $f(d)$ for any $d \in D$. By the strongly fair (resp. weakly fair, proper) scheduler, we mean a strongly fair (resp. weakly fair, proper) scheduler/predicate that is the largest among all strongly fair (resp. weakly fair, proper) schedulers. For example, by definition every execution that satisfies the strongly fair scheduler/predicate also satisfies the weakly fair constraint, but the controversy is not true. So, we consider the weakly fair scheduler/predicate that matches all executions satisfying the weakly fair constraint.

Given a problem, let $\mathcal{SP}$ be its *specification*, which is a predicate defined over the executions of $\mathcal{S}$ and specifies which executions solve the problem. We say that a system $\mathcal{S}$ under a scheduler σ solves the problem if $\mathcal{E}(\mathcal{S}, \sigma) \neq \emptyset$ and $\mathcal{SP}(\xi) = \textbf{true}$ for any $\xi \in \mathcal{E}(\mathcal{S}, \sigma)$. Let σ_W, σ_F and σ_S be the weakly fair, the strongly fair and the synchronous schedulers, respectively. By definition, for any distributed system $\mathcal{S}$, $\mathcal{E}(\mathcal{S}, \sigma_S) \subseteq \mathcal{E}(\mathcal{S}, \sigma_F) \subseteq \mathcal{E}(\mathcal{S}, \sigma_W)$. Hence any problem that cannot be solved under σ_S cannot be solved under σ_F and σ_W. In contrast, any system that can solve a problem under σ_W can solve the problem under σ_F and σ_S.[3] Note that σ_F and σ_W may not be proper and can activate no process despite that there are enabled processes.[4]

2.3 Stabilizing Systems

We now formally define the three notions of stabilizing system used in this paper.

[3]By the same reason, any problem that can be solved under σ_W (resp. σ_F) can be solved under any weakly fair (resp. strongly fair) scheduler, which is the reason we selected σ_W (resp. σ_F) as the representative of the weakly fair (resp. strongly fair) family of schedulers.

[4]In some literature, strongly fair and weakly fair schedulers are assumed to be proper.

Let $\mathcal{S}, \sigma$ and $\mathcal{SP}$ be a distributed system, a scheduler and a specification, respectively.

Definition 1 (Deterministic Self-Stabilization [2]) *$\mathcal{S}$ is deterministically self-stabilizing for $\mathcal{SP}$ under σ, if there is a non-empty subset $\mathcal{L}$ of $\mathcal{C}$ satisfying:*

Strong Closure Property: *Any execution in $\mathcal{E}(\mathcal{S}, \sigma)$ with an initial configuration in $\mathcal{L}$ always satisfies $\mathcal{SP}$.*

Certain Convergence Property: *Any execution in $\mathcal{E}(\mathcal{S}, \sigma)$ eventually reaches a configuration in $\mathcal{L}$.*

Definition 2 (Deterministic Weak Stabilization [13]) *$\mathcal{S}$ is deterministically weak stabilizing for $\mathcal{SP}$ under σ, if there exists a non-empty subset $\mathcal{L}$ of $\mathcal{C}$ satisfying:*

Strong Closure Property: *Any execution in $\mathcal{E}(\mathcal{S}, \sigma)$ with an initial configuration in $\mathcal{L}$ always satisfies $\mathcal{SP}$.*

Possible Convergence Property: *Starting from any configuration, there is an execution in $\mathcal{E}(\mathcal{S}, \sigma)$ that eventually reaches a configuration in $\mathcal{L}$.*

Definition 3 (Probabilistic Self-Stabilization [5]) *$\mathcal{S}$ is probabilistically self-stabilizing for a $\mathcal{SP}$ under σ, if there is a non-empty subset $\mathcal{L}$ of $\mathcal{C}$ satisfying:*

Strong Closure Property: *Any execution in $\mathcal{E}(\mathcal{S}, \sigma)$ with an initial configuration in $\mathcal{L}$ always satisfies $\mathcal{SP}$.*

Probabilistic Convergence Property: *Any execution in $\mathcal{E}(\mathcal{S}, \sigma)$ eventually reaches a configuration in $\mathcal{L}$ with probability 1.*

A configuration is said to be *legitimate* if it is in $\mathcal{L}$, *illegitimate* otherwise.

2.4 Toy Examples

We now give toy examples of deterministic self-stabilizing, deterministic weak-stabilizing, and probabilistic self-stabilizing systems, respectively.

In all examples, we assume a network of two neighboring processes p_0 and p_1. Each process p_x ($x \in \{0, 1\}$) maintains a single boolean variable $V_{p_x} \in \{0, 1\}$. Hence, we denote the configuration of the system by $\langle V_{p_0}, V_{p_1} \rangle$. The set of all possible configurations is then $\{\langle 0, 0\rangle, \langle 0, 1\rangle, \langle 1, 0\rangle, \langle 1, 1\rangle\}$.

The specification $\mathcal{SP}$ is satisfied if and only if $V_{p_0} = V_{p_1}$ and both variables do not change thereafter. In all examples, the set of legitimate configurations $\mathcal{L}$ is $\{\langle 0, 0\rangle, \langle 1, 1\rangle\}$.

The algorithms we give are uniform. Moreover, $\forall x \in \{0, 1\}$, $p_{\overline{x}} = p_{(x+1) \bmod 2}$, *i.e.*, $p_{\overline{x}}$ is the neighbor of p_x. Finally, we assume a strongly fair scheduler.

The algorithm A consisting in one single action R_a (given below) is deterministically self-stabilizing for $\mathcal{SP}$.

$$R_a :: V_{p_x} \neq V_{p_{\overline{x}}} \wedge V_{p_x} = 0 \rightarrow V_{p_x} := 1$$

The state transition system implied by Algorithm A is given in Figure 1.(1). It is easy to see that the algorithm is self-stabilizing for $\mathcal{SP}$. Indeed, from any illegitimate configuration, the system immediately reaches a legitimate one after the scheduler activates the unique enabled process. Moreover, the set of legitimate configurations is trivially closed, as all legitimate configurations are terminal.

Let B be the algorithm consisting in one single action R_b:

$$R_b :: V_{p_x} \neq V_{p_{\overline{x}}} \rightarrow V_{p_x} := V_{(p_x+1)} \bmod 2$$

The state transition system of Algorithm B is given in Figure 1.(2). This state transition system contains a cycle of illegitimate configurations, and the daemon can enforce the system to loop in this cycle. So, B is

not deterministically self-stabilizing for $\mathcal{SP}$. However it is deterministically weak stabilizing for $\mathcal{SP}$. Indeed, from any illegitimate configuration ($\langle 0,1\rangle$ or $\langle 1,0\rangle$) there is a directed path (of length 1) that leads to a legitimate configuration. Similarly to the previous example, the set of legitimate configurations is trivially closed.

Let C be the algorithm consisting in one single action R_c:

$$R_c :: V_{p_x} \neq V_{p_{\overline{x}}} \rightarrow V_{p_x} := \texttt{Rand}()$$

where `Rand` is a random bit generator, which returns one of the two boolean values with the same probability (*i.e.*, $\frac{1}{2}$).

The state transition system of Algorithm C is the same as the one of B. So, Algorithm C is also weak-stabilizing for $\mathcal{SP}$. Now, it is also probabilistically self-stabilizing for $\mathcal{SP}$. Indeed, each time the scheduler activates one or more enabled processes in an illegitimate configuration, there is a positive probability (actually, probability $\frac{1}{2}$) that the next configuration is legitimate. Hence, we can then conclude that C converges with probability 1.

The probability laws for each non-empty choice of the scheduler, assuming that the configuration is $\langle 0,1\rangle$, are given below (the probability laws *w.r.t.* $\langle 1,0\rangle$ are similar):

Activated proc. / Next conf.	$\langle 0,0\rangle$	$\langle 0,1\rangle$	$\langle 1,0\rangle$	$\langle 1,1\rangle$
p_0	0	$\frac{1}{2}$	0	$\frac{1}{2}$
p_1	$\frac{1}{2}$	$\frac{1}{2}$	0	0
p_0, p_1	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$

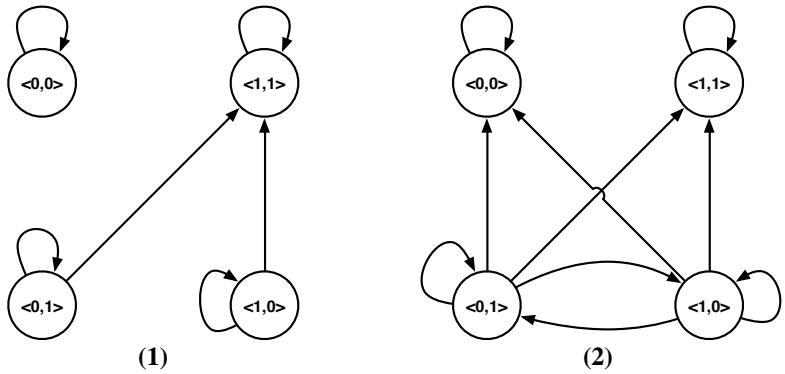

Figure 1: State transition systems of Algorithms A and B: Labeled nodes represent configurations, and arrows denote the possible transitions.

3 Weak Stabilizing Algorithms that are not Self-stabilizing

In this section, we show that weak stabilization is strictly stronger than self-stabilization, from the problem-centric point of view. Intuitively, this result can be understood from the difference of the role of a scheduler in the self-stabilization and the weak stabilization settings. In the former, the scheduler is seen as an *adversary*; the algorithm must stabilize the system, tolerating any "malicious behavior" of the scheduler. Conversely, in the latter, the scheduler can be viewed as a *friend*; the algorithm can expect the scheduler its "most favorite behavior" to stabilize the system. As a matter of fact, the effect of the scheduler is reversed in weak and self-stabilization: the larger the scheduler is (*i.e.*, the more executions are allowed under the scheduler), the easier the weak stabilization can be established, but the harder the self-stabilization is.

In the light of the above intuition, let σ and σ' be any two schedulers such that σ is larger than σ'. We then have (i) if $\mathcal{S}$ is self-stabilizing for $\mathcal{SP}$ under σ, so is $\mathcal{S}$ for $\mathcal{SP}$ under σ', (ii) if $\mathcal{S}$ is self-stabilizing for $\mathcal{SP}$ under σ', $\mathcal{S}$ is weak-stabilizing for $\mathcal{SP}$ under σ'.

An interesting question is the following: Let $\mathcal{S}$ be a weak stabilizing system for $\mathcal{SP}$ under σ'. Is $\mathcal{S}$ also weak stabilizing for $\mathcal{SP}$ under σ? In general, the answer is no. See, for example, the strict clock synchronization problem [16]. There is self-stabilizing, and consequently weak stabilizing, solutions for that problem under the synchronous scheduler. Now, the closure property cannot be always maintained assuming the strongly fair scheduler. However, note that the answer can be yes for a large class of problems, *e.g.*, the silent tasks [17] that are self-stabilizing (resp. weak stabilizing) under σ' are also weak stabilizing under σ.

Finally, note that the converse of (ii) can be true for some schedulers, *e.g.*, we have the following proposition for the synchronous scheduler, since the execution starting from any configuration is uniquely determined.

Proposition 1 *Let $\mathcal{S}$ and $\mathcal{SP}$ be any distributed system and specification, respectively. Under the synchronous scheduler, $\mathcal{S}$ is deterministically weak stabilizing for $\mathcal{SP}$, if and only if it is deterministically self-stabilizing for $\mathcal{SP}$.*

Hence, even if it seems natural that weak stabilization is strictly stronger than self-stabilization from the problem-centric point of view, the result is not trivial. The following two subsections give very simple weak stabilizing (deterministic) algorithms to solve the token passing and the leader election problems on anonymous networks. This establishes that weak stabilization is strictly stronger than self-stabilization, as no deterministically self-stabilizing algorithms can exist for these problems.

3.1 Token Circulation

This subsection considers the token circulation problem in an anonymous unidirectional ring under the strongly fair scheduler. Let $G = (V, E)$ be a unidirectional ring of size N, where $V = \{0, 1, \ldots, N-1\}$ and $E = \{(i, i+1) : i = 0, 1, \ldots, N-1\}$. Note that the name i of process is calculated modulo N. Variables in process i are hence readable only from process (i and) $i+1$.

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

There is no deterministic self-stabilizing token circulation algorithm on an anonymous unidirectional ring under the strongly fair scheduler [7].

A Weak Stabilizing Algorithm. Algorithm 1, called $\mathcal{TC}$ in the following, is the $(N-1)$-fair algorithm proposed by Beauquier et al. [18]. It is $(N-1)$-fair, in the sense that (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, regardless of the scheduler under which it is executed. We show that it is a weak stabilizing token circulation algorithm under the strongly fair scheduler. Let m_N be the smallest integer not dividing N.[5]

In Algorithm $\mathcal{TC}$, each process i maintains a single variable v_i whose domain is $\{0, \ldots, m_N - 1\}$. This variable allows i to know if it holds the token or not. We actually define that a process i holds a token, if and only if $v_i \neq ((v_{i-1} + 1) \bmod m_N)$, *i.e.*, if and only if $Token_i$ is true at i. In this case, Action `A` is enabled at i. This action allows i to pass the token to $i+1$.

Figure 2 depicts an execution of Algorithm $\mathcal{TC}$ starting from a legitimate configuration, *i.e.*, a configuration such that there is exactly one process holding the token. In this example, $N = 6$ and $m_N = 4$. In each configuration, the process with an asterisk is the token holder. By executing Action `A`, it passes the token to its successor. It is indeed obvious to observe the following proposition by the definition of Algorithm $\mathcal{TC}$.

[5]Process i does not know its name i, and notation v_i in Algorithm $\mathcal{TC}$ does not tell process i its name i; it should read as v, and its index i is only for the convenience of readers. Since $\delta_i^- = 1$, v_{i-1} should read as v_0 (see the model of anonymous distributed system in Section 2), *i.e.*, the v of the predecessor (of process i) whose local index is 0; index $i-1$ again is not visible from process i. In this algorithm, process i has exactly one predecessor $i-1$, and v_0 for process i is hence uniquely determined. The notation v_{i-1} is again only for the convenience of readers.

Algorithm 1 Algorithm $\mathcal{TC}$, code for every process i

Variable: $v_i \in \{0, \ldots, m_N - 1\}$
Macro:
$PassToken_i \equiv v_i := (v_{i-1} + 1) \bmod m_N$
Predicate:
$Token_i \equiv [v_i \neq ((v_{i-1} + 1) \bmod m_N)]$
Action:
A :: $Token_i \rightarrow PassToken_i$

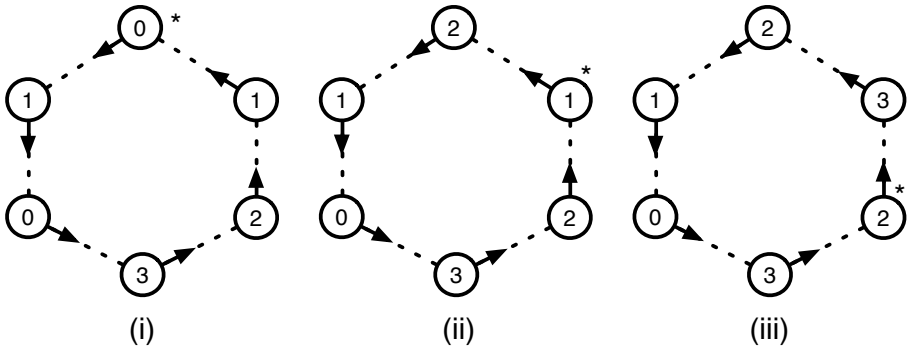

Figure 2: An execution of Algorithm $\mathcal{TC}$ starting from a legitimate configuration.

Proposition 2

1. *A process is enabled if and only if it has a token; only a token holder is activated.*
2. *Suppose that at some time instant t process i is a token holder but process $i+1$ is not. If i is activated at t, then i loses a token and $i+1$ instead has a token at $t+1$.*

Let $\mathcal{C}$ be the set of configurations. For any configuration $\gamma \in \mathcal{C}$, the set of processes i holding a token is denoted by $\mathtt{TH}(\gamma)$. We call a configuration γ legitimate if $|\mathtt{TH}(\gamma)| = 1$, and let $\mathcal{L} = \{\gamma \in \mathcal{C} : |\mathtt{TH}(\gamma)| = 1\}$ be the set of legitimate configurations.

The specification $\mathcal{SP}$ of this problem can be defined as follows: For any execution $\xi = \gamma_0, \gamma_1, \ldots$, $\mathcal{SP}(\xi) =$ **true** if $\gamma_i \in \mathcal{L}$ for all $i \geq 0$. To verify that $\mathcal{SP}$ indeed specifies the token circulation, observe that every process has a token infinitely often in any execution $\xi \in \mathcal{SP}(\xi)$ by Proposition 2.

We show that Algorithm $\mathcal{TC}$ is a deterministically weak stabilizing token circulation algorithm under the strongly fair scheduler. More formally, let $\mathcal{S}$ be the distributed system executing Algorithm $\mathcal{TC}$ on an unidirectional anonymous ring G of size N. We show that $\mathcal{S}$ is deterministically weak stabilizing for $\mathcal{SP}$ under the strongly fair scheduler.

Let i and j be two distinct processes. We denote by $P(i, j)$ the unique (simple) path from i to j. Note that $P(i, j) \neq P(j, i)$. Recall that $d(i, j)$ denotes the distance from i to j, *i.e.*, $d(i, j) = |P(i, j)| - 1$.

Lemma 1 $|\mathtt{TH}(\gamma)| > 0$ *for any configuration* $\gamma \in \mathcal{C}$.

Proof. To derive a contradiction, assume that there is a configuration γ such that $|\mathtt{TH}(\gamma)| = 0$. By definition, $v_i = (v_{i-1} + 1) \bmod m_N$ for all $i = 0, 1, \ldots, N-1$, and hence $v_0 = (v_0 + N) \bmod m_N$, a contradiction, since m_N does not divide N. □

Lemma 2 *For any configuration $\gamma \in \mathcal{C}$, there is a legitimate configuration $\gamma' \in \mathcal{L}$ reachable from γ under the strongly fair scheduler.*

Proof. Since $|\mathtt{TH}(\gamma)| \geq 1$ for any configuration $\gamma \in \mathcal{C}$ by Lemma 1, it suffices to show that for any configuration $\gamma \in \mathcal{C}$ such that $|\mathtt{TH}(\gamma)| > 1$, there is a configuration $\gamma' \in \mathcal{C}$ such that it is reachable from γ under the strongly fair scheduler and that $|\mathtt{TH}(\gamma')| < |\mathtt{TH}(\gamma)|$ holds.

Let i and j be two consecutive (clockwise) token holders and assume that a configuration γ' is obtained as the result of an activation of process i. Then $\mathtt{TH}(\gamma') \subseteq \mathtt{TH}(\gamma) \setminus \{i\}$ and $|\mathtt{TH}(\gamma')| \leq |\mathtt{TH}(\gamma)| - 1$ if $j = i + 1$, and $\mathtt{TH}(\gamma') = (\mathtt{TH}(\gamma) \setminus \{i\}) \cup \{i + 1\}$ if $j > i + 1$, by Proposition 2. An execution (segment) that sequentially activates $j - i$ processes $i, i + 1, \ldots, j - 1$ in this order one by one thus decreases the number of tokens by at least 1. Note that this prefix of execution obviously does not violate the strong fairness. □

The possible convergence is obtained by Lemma 2. The strong closure simply follows from Proposition 2. Moreover, once stabilized, every process has the token infinitively often, implying that the strong fairness is never violated. Hence, we can conclude with the next theorem:

Theorem 1 *Algorithm $\mathcal{TC}$ is a deterministically weak stabilizing token passing algorithm under the strongly fair scheduler.*

3.2 Leader Election

We next consider the leader election problem on an anonymous undirected tree under the strongly fair scheduler.

Definition 5 (Leader Election) *The leader election problem is the problem of distinguishing a unique process as a* leader *in the network in such a way that the distinguished process knows that it is the leader and all other processes know that they are not the leader.*

Following the results of Yamashita and Kameda [19], there is an anonymous tree whose symmetricity is 2 and deterministic leader election is impossible on such a tree under the synchronous scheduler. Hence there is no (deterministic) self-stabilizing leader election algorithm on an anonymous tree under the strongly fair scheduler. The following simple example verifies this fact.

Let $\mathcal{A}$ be any leader election algorithm on an anonymous tree. Consider any execution of $\mathcal{A}$ on an anonymous chain consisting of two processes p_0 and p_1 under the synchronous scheduler. Let $\overline{\mathcal{L}}$ be the set of configurations such that the local states of both processes are the same. Any $\gamma \in \overline{\mathcal{L}}$ is not legitimate by definition. Since the scheduler is synchronous, $\gamma \in \overline{\mathcal{L}}$ and $\gamma \mapsto \gamma'$ implies $\gamma' \in \overline{\mathcal{L}}$ because of the symmetry between p_0 and p_1. Hence $\mathcal{A}$ does not elect a leader if its initial configuration is selected from $\overline{\mathcal{L}}$, *i.e.*, $\mathcal{A}$ is not a self-stabilizing leader election algorithm under the synchronous (and therefore the strongly fair) scheduler.

A weak stabilizing algorithm for the same problem under the strongly fair scheduler is, to the contrary, easy to construct from the self-stabilizing algorithm for finding the centers of a tree presented in [20]. Recall that in a tree there is a unique center or there are two neighboring centers by Property 1. The algorithm defines a predicate *Center* over its local state, and guarantees that its execution starting from any configuration eventually reaches a configuration such that *Center* is true at a process if and only if it is a center of the tree. Furthermore, a processes can locally decide if *Center* is true at one of its neighbors.

We add the following algorithm segment to the algorithm: If a process is a center, but none of its neighbors are, then it considers itself as the leader. In order to break a tie when the tree has neighboring centers, a boolean variable B is prepared at each process, and we allow a center to change the value of B anytime until it is different of the other center. If the B values of the centers are different, the center with $B =$ **true** then considers itself as the leader. Since there is a transition that changes the B value of exactly one of the centers, the modified algorithm is obviously a weak stabilizing leader election algorithm under the strongly fair scheduler.

This algorithm however uses $\log N$ bits of memory per process. We show that this space complexity is reducible to 2 bits in average per process, by presenting a new algorithm called Algorithm $\mathcal{LE}$ (its code is given in Algorithm 2).

Algorithm $\mathcal{LE}$ maintains in each process $p \in V$ a single variable v_p whose domain is $L_p \cup \{\perp\}$. Recall that p identifies its δ_p neighbors by the local indices $L_p = \{0, 1, \ldots, \delta_p - 1\}$. We assume that each neighbor

q of p knows its local index i_q in L_p. Hence, q can test if $v_p = i_q$ (in the following we denote such a test by $v_p = q$ for simplicity). Process p considers itself as the leader, if and only if $v_p = \perp$. If $v_p \neq \perp$, v_p points the *parent* of p, and p is said to be a *child* of its parent. Algorithm $\mathcal{LE}$ at process p hence uses $\lceil \log(\delta_p + 1) \rceil$ bits of memory. Since $\delta_p \geq 1$, the total number of bits that Algorithm $\mathcal{LE}$ uses is

$$\sum_{p \in V} \lceil \log(\delta_p + 1) \rceil \leq \sum_{p \in V} (1 + \log \delta_p).$$

Since $\sum_{p \in V} \delta_p \leq 2(n-1)$,

$$\frac{\sum_{p \in V} \log \delta_p}{n} = \log \sqrt[n]{\prod_{p \in V} \delta_p} \leq \log \frac{\sum_{p \in V} \delta_p}{n} \leq \log 2 = 1.$$

Thus the average number of bits that Algorithm $\mathcal{LE}$ uses per process is at most 2.

Algorithm 2 Algorithm $\mathcal{LE}$, code for any process p

Variable: $v_p \in L_p \cup \{\perp\}$
Macro:
$Child_p \equiv \{q \in L_p : v_q = p\}$
Predicates:
$Leader_p \equiv (v_p = \perp)$
Actions:

$\mathtt{A}_1$	::	$(v_p \neq \perp) \wedge (\lvert Child_p \rvert = \delta_p)$	$\rightarrow$	$v_p := \perp$
$\mathtt{A}_2$	::	$(v_p \neq \perp) \wedge [L_p \setminus (Child_p \cup \{v_p\}) \neq \emptyset]$	$\rightarrow$	$v_p := (v_p + 1) \bmod \delta_p$
$\mathtt{A}_3$	::	$(v_p = \perp) \wedge (\lvert Child_p \rvert < \delta_p)$	$\rightarrow$	$v_p := \min(L_p \setminus Child_p)$

Let γ be a configuration. The directed graph $T_\gamma = (V, A_\gamma)$ associated with γ is defined as follows: For each $p \in V$, if q is the parent of p then $(p, q) \in A_\gamma$. Let $\delta_p^+(T_\gamma)$ be the outdegree of a process $p \in V$ in T_γ. By definition $\delta_p^+(T_\gamma) \leq 1$ for each process $p \in V$. Algorithm $\mathcal{LE}$ then tries to reach a terminal configuration γ such that T_γ is a rooted intree. An intuitive explanation of each action is as follows:

($\mathtt{A}_1$) If a process p such that $v_p \neq \perp$ is pointed as the parent by all of its neighbors, then p sets v_p to $\perp$ and starts to consider itself as the leader.

($\mathtt{A}_2$) If a process p such that $v_p \neq \perp$ has a neighbor that is neither its parent nor one of its children, which means that not all processes among p and its neighbors consider the same process as the leader, p changes its parent by incrementing its parent pointer modulo δ_p.

($\mathtt{A}_3$) If a process p satisfies $v_p = \perp$ but at least one of its neighbors q is not its child, which means that q considers another process as the leader, p stops to consider itself as the leader by pointing, by v_p, one of its non-child neighbors.

Figure 3 depicts an execution of Algorithm $\mathcal{LE}$ that converges. The circles and the dashed lines respectively represent the processes and the bidirectional communication links. The labels of processes are only for the purpose of explanation. An arrow from a process p to another process q shows that q is the parent of p, *i.e.*, $v_p \in L_p$ is the local index of q for p. A label $\mathtt{A}_\mathtt{j}$ attached to a process p means that $\mathtt{A}_\mathtt{j}$ is enabled at p. When the processes with labels $\mathtt{A}_\mathtt{j}$ with asterisks in a figure are executed, the configuration shown in the next figure is reached. Observe that the directed graph T_γ in (v) is a rooted intree.

For example, in the configuration in (i), no process p satisfies $v_p = \perp$ and considers itself as the leader, but p_1, p_2, p_7, and p_8 are pointed by all of their respective neighbors. They are candidates to become the leader (Action $\mathtt{A}_1$). Also p_3, p_5, and p_6 are enabled to execute Action $\mathtt{A}_2$: They have a neighbor that is neither their parent nor one of their children. Finally, p_4 is not enabled. For configuration γ in (v), T_γ is a rooted intree and hence γ is a terminal configuration.

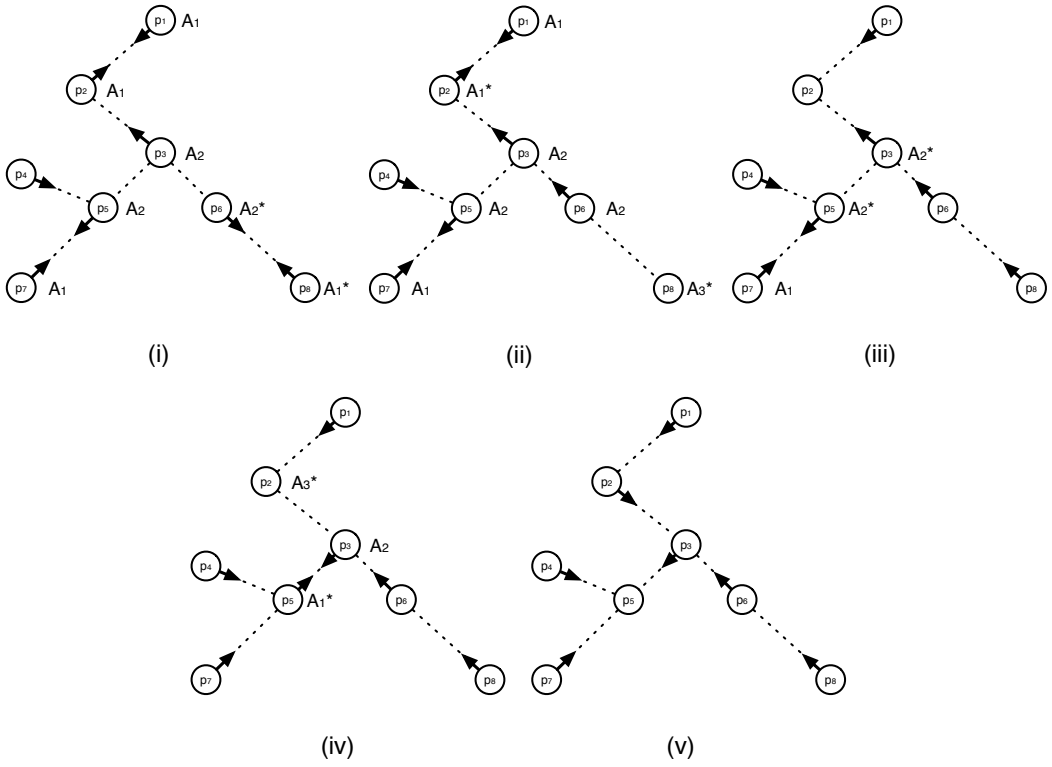

Figure 3: An execution of Algorithm $\mathcal{LE}$ that converges.

Let $\mathcal{C}$ be the set of all configurations. A configuration $\gamma \in \mathcal{C}$ is legitimated if T_γ is a rooted intree. Note that there is exactly one leader, who is the root, in a legitimate configuration. Let $\mathcal{L}$ be the set of all legitimate configurations. We then define the specification $\mathcal{SP}$ of this problem as follows: for any execution $\xi = \gamma_0, \gamma_1, \ldots$, $\mathcal{SP}(\xi) = \textbf{true}$ if $\gamma_j = \gamma_i \in \mathcal{L}$ for all $j \geq 0$. We now show that Algorithm $\mathcal{LE}$ is a deterministically weak stabilizing algorithm for $\mathcal{SP}$ under the strongly fair scheduler.

Lemma 3 *Let $\gamma \in \mathcal{C}$ be any configuration. If there is no leader in γ,* i.e.*, all processes have outdegree 1 in T_γ, there is a process p such that Action* $\mathtt{A_1}$ *is enabled at p.*

Proof. Let $G = (V, E)$ be an (undirected) tree that represents the communication network of the distributed system. We first show that there is a process $p \in V$ such that $\delta_p = \delta_p^-(T_\gamma)$ by induction on $|V|$, where δ_p and $\delta_p^-(T_\gamma)$ are respectively the degree of p in G and the indegree of p in T_γ.

Since the case $|V| = 1$ is trivial, we take the case $|V| = 2$ as the base case. Let $V = \{p, q\}$. Since $\delta_q^+(T_\gamma) = 1$, $(q, p) \in A_\gamma$, which implies that p satisfying the condition, since $\delta_p = 1$.

For induction step, without loss of generality, we may assume that T_γ is weakly connected. Let $p \in V$ be a leaf of G and let $q \in V$ be its neighbor. If $(q, p) \in A_\gamma$ then p is a process satisfying the condition.

If $(q, p) \notin A_\gamma$, since $(p, q) \in A_\gamma$ (because $\delta_p^+(T_\gamma) = 1$), consider a tree $G' = (V \setminus \{p\}, E \setminus \{(p, q)\})$ and a directed acyclic graph $T'_\gamma = (V \setminus \{p\}, A_\gamma \setminus \{(p, q)\})$. By applying the induction hypothesis to G' and T'_γ, there is a process $r \in V \setminus \{p\}$ such that $\delta_r(G') = \delta_r^-(T'_\gamma)$, where $\delta_r(G')$ is the degree of r in G'.

If $r \neq q$, then $\delta_r(G') = \delta_r$, and r is a process (in G) satisfying the condition. If $r = q$, then again r is a process (in G) satisfying the condition, since $(p, q) \in A_\gamma$.

Let p be a process in G satisfying the condition. Since $v_p \neq \perp$, Action $\mathtt{A_1}$ is enabled at p. □

By Lemma 3, the following corollary holds.

Corollary 1 *Let $\gamma \in \mathcal{C}$ be any configuration. If there is no leader in γ, then from γ after one step, a configuration is reached such that there is a leader.*

Lemma 4 *A legitimate configuration is reachable, under the strongly fair scheduler, from any configuration such that there is a leader.*

Proof. Consider any configuration $\gamma \in \mathcal{C}$ and suppose that *Leader* holds at a process p. Let $H = (U, A)$ be the (maximum) sub-intree of T_γ rooted at p. If $U = V$, then $T_\gamma (= H)$ is a rooted intree and hence $\gamma \in \mathcal{L}$.

Suppose that $U \subset V$ and let $(q, r) \in E$ be an edge of G that connects two nodes, one in U and the other in $V \setminus U$, *i.e.*, $(q, r) \in U \times (V \setminus U)$. By definition neither (q, r) nor (r, q) are in A_γ. It suffices to show that there is an execution from γ to change the value of v_r to point to q (without changing the other v values).

If $v_r = \perp$, Action $\mathtt{A_3}$ is enabled at r, since $(q, r) \notin A_\gamma$. Without loss of generality, we hence assume that $v_r \neq \perp$ in γ. Action $\mathtt{A_2}$ is then enabled at r. By activating r several times, we can set v_r to point to q.

It is easy to observe that this execution (segment) does not violate the strong fairness. □

Theorem 2 *Algorithm $\mathcal{LE}$ is a deterministically weak stabilizing algorithm for $\mathcal{SP}$ under the strongly fair scheduler.*

Proof. By Corollary 1 and Lemma 4, the possible convergence property holds. The strong closure property also holds, since a legitimate configuration is a terminal configuration. Moreover, since a terminal configuration is reached, the execution satisfies the strongly fair scheduler. □

4 From Weak to Probabilistic Stabilization

Gouda showed that deterministic weak stabilization is a "good approximation" of deterministic self-stabilization by proving the following theorem [13]:[6]

Theorem 3 ([13]) *Any deterministic weak stabilizing system under the strongly fair scheduler is also a deterministic self-stabilizing system if*

1. *the system has a finite number of configurations, and*
2. *the scheduler satisfies the Gouda's strong fairness assumption.*

An algorithm is said to be *finite state* if it makes use of finite memory, *i.e.*, the domain of each of the variables is finite. Since a distributed system consists of a finite number of processes, the number of configurations is finite if and only if the algorithm is finite state, which is an important assumption in the first two subsections (as well as Theorem 3). The last subsection then considers infinite state algorithms.

From Theorem 3, one may conclude that deterministic weak stabilization and deterministic self-stabilization are equivalent, under the strongly fair scheduler. This would contradict the results presented in Section 3. Actually, this is not the case: we show that the Gouda's strong fairness assumption is *in essence* strictly stronger than the strong fairness.

Let us consider an algorithm such that every process is enabled in any configuration and each of the actions does not change the local state. By definition $\xi = \gamma_0, \gamma_0, \ldots$ is the only execution under any scheduler, for any initial configuration γ_0. Any scheduler (which may not be fair in some sense) thus satisfies the Gouda's strong fairness assumption. In this sense, the Gouda's strong fairness is in general incomparable with the strong or the weak fairness. We introduce a natural assumption to avoid this pathological situation and assume it in Section 4; an action always changes the local state by writing a new value to a local variable.

Theorem 4 *Suppose that the system has a finite number of configurations. The Gouda's strong fairness is strictly stronger than the strong fairness.*

[6] This result has been proven assuming a scheduler that is central. However, it is easy to see that the proof given in [13] still holds if we remove this constraint.

Proof. Suppose that a process p is enabled infinitely many times in some execution ξ. We show that p is activated infinitely many times in ξ, provided that the scheduler satisfies the Gouda's strong fairness.

Since the number of configurations is finite, in ξ, p is enabled infinitely many times in a configuration γ. Then there is another configuration $\gamma'(\neq \gamma)$ that is reached from γ by the activation of p. That is, $\gamma \mapsto \gamma'$. Thus $\gamma \mapsto \gamma'$ occurs infinitely many times in ξ, if the scheduler satisfies the Gouda's strong fairness, which implies that p is activated infinitely many times.

We next observe that the Gouda's strong fairness is strictly stronger than the strong fairness. Let $\mathcal{S}$ be the distributed system executing Algorithm $\mathcal{TC}$ on an anonymous unidirectional ring. Since there is no deterministically self-stabilizing token circulation algorithm on an anonymous unidirectional ring under the strongly fair scheduler [7], $\mathcal{S}$ is not deterministically self-stabilizing under the strongly fair scheduler. However, $\mathcal{S}$ is deterministically self-stabilizing under any scheduler satisfying the Gouda's strong fairness, since it is deterministically weak stabilizing under the strongly fair scheduler. Thus the Gouda's strong fairness is strictly stronger than the strong fairness. □

4.1 Finite State Deterministic Algorithm under Randomized Scheduler

Let $\mathcal{S}$ be a (possibly randomized) distributed system. Suppose that $\mathcal{S}$ is in a configuration γ and let $U \subseteq V$ be the set of enabled processes in γ. Then a subset $W \subseteq U$ is selected uniformly at random from 2^U by the randomized scheduler, and the next configuration is determined from W. We model the distributed system $\mathcal{S}$ under the randomized scheduler as a finite state Markov chain $\mathcal{M}$ defined over the set $\mathcal{C}$ of "states" associated with the transition probability matrix $\mathbf{P} = (p_{\gamma\gamma'})_{\gamma,\gamma' \in \mathcal{C}}$, where the transition probability $p_{\gamma\gamma'}$ from γ to γ' is $2^{-|U|}$ by the definition of the randomized scheduler. The execution ξ with initial configuration $\gamma_0 \in \mathcal{C}$ (as a stochastic process) is the evolution of $\mathcal{M}$, which is a sequence $x_0, x_1, \ldots$ of random variables x_t representing the configuration at time instant t, such that

$$\Pr(x_0 = \gamma_0) = 1,$$

and for $t = 0, 1, \ldots$,

$$\Pr(x_{t+1} = \gamma_{t+1} | x_t = \gamma_t, x_{t-1} = \gamma_{t-1}, \ldots, x_0 = \gamma_0) = \Pr(x_{t+1} = \gamma_{t+1} | x_t = \gamma_t) = p_{\gamma_t \gamma_{t+1}}.$$

We call $\mathcal{M}$ the Markov chain corresponding to $\mathcal{S}$ (under the randomized scheduler).

Since we analyze finite Markov chains in this and the next subsections, and infinite Markov chains in Subsection 4.3, here we introduce concepts both for finite and infinite Markov chains. A (finite or infinite) Markov chain is called *irreducible* if its state transition graph is strongly connected. A state γ is called *recurrent* if the the evolution $x_0, x_1, \ldots$ that starts in γ returns γ with probability 1, and otherwise it is called *transient*.

Let $\mathcal{C}_1, \mathcal{C}_2, \ldots$ be the strongly connected components of $\mathcal{S}$. The *component graph* H of $\mathcal{S}$ is constructed from $\mathcal{S}$ by contracting each strongly connected components; *i.e.*, $H = (\{v_1, v_2, \ldots\}, A)$, where $(v_i, v_j) \in A$ if and only if $i \neq j$ and there are configurations $\gamma \in \mathcal{C}_i$ and $\gamma' \in \mathcal{C}_j$ such that $\gamma \mapsto \gamma'$. H is a (finite or infinite) directed acyclic graph (DAG). A node v of H is called a *source* (resp. *sink*) if $\delta_v^-(H) = 0$ (resp. $\delta_v^+(H) = 0$), where $\delta_v^-(H)$ (resp. $\delta_v^+(H)$) is the indegree (resp. outdegree) of node v in H. Let D be the set of sinks in H.

Theorem 5 *Let $\mathcal{S}$ be the distributed system executing a finite state deterministic algorithm $\mathcal{A}$ on a communication network G, and let $\mathcal{SP}$ be any specification for $\mathcal{S}$. Then, $\mathcal{S}$ is deterministically weak stabilizing for $\mathcal{SP}$ under the strongly fair scheduler, if and only if it is probabilistically self-stabilizing for $\mathcal{SP}$ under the randomized scheduler.*

Proof. Since the *If* part is obvious by definition, we concentrate on the *Only-If* part. Suppose that $\mathcal{S}$ is weakly self-stabilizing for $\mathcal{SP}$ under the strongly fair scheduler. Obviously, $\mathcal{S}$ holds the strong closure property under the randomized scheduler.

As for the probabilistic convergence property, let $\mathcal{M}$ be the Markov chain corresponding to $\mathcal{S}$. For any configuration $\gamma_0 \in \mathcal{C}$, consider the evolution $x_0, x_1, \ldots$ of $\mathcal{M}$ starting in γ_0. Without loss of generality, we may assume that all configurations are reachable from γ_0.

Let $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k$ be the strongly connected components of $\mathcal{S}$ and consider the component graph $H = (\{v_1, v_2, \ldots, v_k\}, A)$ of $\mathcal{S}$. H is a finite DAG. Since all configurations are assumed to be reachable from γ_0, H has the single source, say v_1, and $\gamma_0 \in \mathcal{C}_1$. If $v_i \in D$, $\mathcal{C}_i$ must contain a legitimate configuration, since $\mathcal{S}$ is deterministically weak stabilizing.

For any j such that $v_j \notin D$, $|\mathcal{C}_j| < \infty$ since $\mathcal{S}$ is finite state, and all configurations in $\mathcal{C}_j$ are transient.

Hence the evolution $x_0, x_1, \ldots$ eventually reaches, with probability 1, a component $\mathcal{C}_i$ such that $v_i \in D$ (see *e.g.*, [21]). Since all configurations in $\mathcal{C}_i$ are recurrent, it eventually visits, with probability 1, all configurations in $\mathcal{C}_i$, including the legitimate configuration (see *e.g.*, [21]). □

Recall that the randomized scheduler selects and activates each enabled process with probability 1/2. We can easily extend the above theorem to a randomized scheduler that selects and activates each enabled process with probability c for any $0 < c < 1$. Nevertheless, the theorem cannot be extended to $c = 1$, since one can easily observe that Algorithm $\mathcal{LE}$ is not a self-stabilizing algorithm under the synchronous scheduler. It is however desirable to have a solution that works under the synchronous scheduler, since it is easy to implement. This is the focus of the next subsection.

4.2 Finite State Randomized Algorithm under Synchronous Scheduler

We propose a simple transformer that transforms a finite state deterministically weak self-stabilizing algorithm $\mathcal{A}$ into a randomized algorithm $\mathcal{A}^*$, borrowing an idea from [6]: Whenever an enabled process is activated by the synchronous scheduler, it first tosses a coin and then executes the action, if and only if it gets the "head". If the system $\mathcal{S}$ executing $\mathcal{A}$ on a communication network G is weakly stabilizing for a specification $\mathcal{SP}$ under the strongly fair scheduler, we show that the system $\mathcal{S}^*$ executing $\mathcal{A}^*$ on G is probabilistically self-stabilizing for a specification $\mathcal{SP}^*$, which is *essentially* the same as $\mathcal{SP}$, under the synchronous scheduler.

In $\mathcal{A}^*$, in addition to the variables used in $\mathcal{A}$, we provide a new boolean variable B. We then replace each action $\mathtt{A} :: Guard_{\mathtt{A}} \rightarrow \mathtt{S_A}$ of $\mathcal{A}$ into the following action $\mathtt{Trans(A)}$:

$$\mathtt{Trans(A)} :: Guard_{\mathtt{A}} \rightarrow B := \mathtt{Rand}(); \textbf{ if } B \textbf{ then } \mathtt{S_A},$$

where `Rand` is a random bit generator, which returns one of the boolean values with the same probability.

We continue to use the notations in the last subsection and let us model $\mathcal{S}$ under the randomized scheduler by the Markov chain $\mathcal{M}$. We also model $\mathcal{S}^*$ under the synchronous scheduler as another finite state Markov chain $\mathcal{M}^*$ defined over the set $\mathcal{C}^*$ of configurations of $\mathcal{S}^*$ associated with the transition probability matrix $\mathbf{P}^*$.

Let γ^* be any configuration in $\mathcal{C}^*$. If we ignore the values of B_ps for all $p \in V$ in γ^*, we obtain a configuration in $\mathcal{C}$, which we denote by γ. The correspondence between γ and γ^* is one-to-many.

Consider any configuration γ_1 and let U be the set of enabled processes in γ_1. If γ_2 is reached from γ_1 by activating a set $W \subseteq U$ of processes, then the transition probability from γ_1 to γ_2 is $2^{-|U|}$, as discussed in the previous subsection.

By definition, U is also the set of enabled processes in any configuration $\gamma_1^* \in \mathcal{C}^*$. Then the synchronous scheduler activates all processes in U. However only processes $p \in U$ that have $B_p = \textbf{true}$ proceed to execute their actions. If $W \subseteq U$ is the set of processes that have proceeded to their actions, a configuration $\gamma_2^* \in \mathcal{C}^*$ is reached. It is worth emphasizing that γ_2^* is uniquely determined from the values of B_ps in γ_1^*, although the correspondence between γ_2 and γ_2^* is in general one to many, and that the probability that W is selected is by definition $2^{-|U|}$. There is thus a one to one correspondence between sampling paths $\xi = \gamma_0, \gamma_1, \ldots$ of $\mathcal{M}$ and $\xi^* = \gamma_0^*, \gamma_1^*, \ldots$ of $\mathcal{M}^*$, and their evolutions follow the same probability distribution.

Define $\mathcal{SP}^*$ by $\mathcal{SP}^*(\xi^*) = \mathcal{SP}(\xi)$, and let $\mathcal{L}^* = \{\gamma^* : \gamma \in \mathcal{L}\}$. Then the above observation implies that $\mathcal{S}$ under the randomized scheduler holds the probabilistic convergence property for $\mathcal{SP}$, if and only if $\mathcal{S}^*$ under

the synchronous scheduler holds it for $\mathcal{SP}^*$. It is also obvious that $\mathcal{S}$ holds the strong closure property for $\mathcal{SP}$ under the randomized scheduler, if and only if $\mathcal{S}^*$ holds it for $\mathcal{SP}^*$ under the synchronous scheduler.

Theorem 6 *$\mathcal{S}$ is probabilistically self-stabilizing for $\mathcal{SP}$ under the randomized scheduler, if and only if $\mathcal{S}^*$ is probabilistically self-stabilizing for $\mathcal{SP}^*$ under the synchronous scheduler.*

4.3 Infinite State Deterministic Algorithm under Randomized Scheduler

We have shown that a finite state algorithm is deterministically weak stabilizing under the strongly fair scheduler, if and only if it is probabilistically self-stabilizing under the randomized scheduler. This subsection considers infinite state algorithms.

Let $\mathcal{A}$ be an infinite state algorithm that is deterministically weak stabilizing for a specification $\mathcal{SP}$ under the strongly fair scheduler. Consider the distributed system $\mathcal{S} = (\mathcal{C}, \mapsto)$ executing $\mathcal{A}$ on a communication graph G under the randomized scheduler, and let $\mathcal{M}$ be the Markov chain corresponding to $\mathcal{S}$; $\mathcal{M}$ is defined over $\mathcal{C}$ associated with the transition probability matrix $\mathbf{P} = (p_{\gamma\gamma'})_{\gamma,\gamma' \in \mathcal{C}}$, where for any configurations $\gamma, \gamma' \in \mathcal{C}$ such that $\gamma \mapsto \gamma'$, $p_{\gamma\gamma'} = 2^{-|U|}$. Here U is the set of processes enabled in γ. Since $\mathcal{A}$ is an infinite state algorithm, $\mathcal{M}$ is an infinite Markov chain.

Like finite algorithms, $\mathcal{S}$ holds the strong closure property for a specification $\mathcal{SP}$ under the strongly fair scheduler, if and only if it holds the same property for $\mathcal{SP}$ under the randomized scheduler. Also, $\mathcal{S}$ holds the possible convergence property holds for $\mathcal{SP}$ under the strongly fair scheduler, if it holds the probabilistic convergence property for $\mathcal{SP}$ under the randomized scheduler. However, its converse may not hold in general, unlike finite state algorithms.

Let us consider the following simple algorithm $\mathcal{A}$ on a distributed system consisting only of 3 processes p, q and r. A process provides a variable v, whose domain is the set of non-negative integers. $\mathcal{A}$ consists of a single rule `A` to increment v.

$$\texttt{A} :: \mathbf{true} \to v := v + 1.$$

Since the guard is **true**, all processes are always enabled. We call a configuration legitimate if $v_p + v_q = v_r$. Let $\mathcal{L}$ be the set of all legitimate configurations. Finally define a specification $\mathcal{SP}$ by $\mathcal{SP}(\xi) = \mathbf{true}$ for any ξ that contains a legitimate configuration.

Then this distributed system obviously holds the possible convergence property for $\mathcal{SP}$ under the strongly fair scheduler, but does not hold the probabilistic convergence property for $\mathcal{SP}$ under the randomized scheduler, since the corresponding Markov chain $\mathcal{M}$ is (a variation of) an asymmetric unrestricted random walk on the line, which is known to be irreducible and transient [22, Example 8.9]. Infinite state deterministically weak stabilizing systems thus cannot be automatically transformed into probabilistically self-stabilizing systems by replacing the scheduler, unlike finite state deterministically weak stabilizing systems.

Proposition 3 *There is an infinite state algorithm $\mathcal{A}$ that is deterministically weak stabilizing for a specification $\mathcal{SP}$ under the strongly fair scheduler, such that $\mathcal{A}$ is not probabilistically self-stabilizing for $\mathcal{SP}$ under the randomized scheduler.*

Let $\mathcal{SP}$ be a specification defined by $\mathcal{SP}(\xi) = \mathbf{true}$ for all execution ξ. Then any distributed system, including the above $\mathcal{S}$, is both deterministically weak stabilizing under the strongly fair scheduler and probabilistically self-stabilizing under the randomized scheduler, for this $\mathcal{SP}$. That is, the transformability of an infinite state distributed system depends not only on the system but also on the specification, which is a notable difference between finite state and infinite state systems.

However, some of infinite state deterministically weak stabilizing systems are transformed into probabilistically self-stabilizing systems by replacing the scheduler, for *any* specification. If an infinite Markov chain is (irreducible and) recurrent, then the probability that the evolution starting in γ eventually reaches γ' is 1 for any two states γ and γ'. Thus we have the following proposition.

Proposition 4 *Let $\mathcal{S}$ and $\mathcal{M}$ be an infinite state distributed system and the infinite Markov chain corresponding to $\mathcal{S}$, respectively. Assume that $\mathcal{M}$ is (irreducible and) recurrent. Let $\mathcal{SP}$ be any specification for*

$\mathcal{S}$. Then $\mathcal{S}$ is deterministically weak stabilizing for $\mathcal{SP}$ under the strongly fair scheduler, if and only if it is probabilistically self-stabilizing for $\mathcal{SP}$ under the randomized scheduler.

Let $\mathcal{S}$ be distributed system. In the rest of this subsection, we explore weaker sufficient conditions (than the one in Proposition 4) to guarantee that, for any specification $\mathcal{SP}$, $\mathcal{S}$ holds the possible convergence property for $\mathcal{SP}$ under the strongly fair scheduler, if and only if it holds the probabilistic convergence property for $\mathcal{SP}$ under the randomized scheduler.

By Theorem 5, one of such sufficient conditions is that $\mathcal{S}$ is finite state; *i.e.*, the number of strongly connected components and the cardinality of each strongly connected component $\mathcal{C}_i$ are both finite. We extend Theorem 5 in two directions; when the number of strongly connected components $\mathcal{C}_1, \mathcal{C}_2, \ldots$ is infinite, and when the cardinality of some of $\mathcal{C}_i$ is infinite. The next theorem extends Theorem 5 in the first direction.

Recall that we denote by $H = (\{v_1, v_2, \ldots, v_k\}, A)$ the component graph of $\mathcal{S}$, where every node v_i represents the component $\mathcal{C}_i$. Recall that D is the set of sinks in H.

Theorem 7 *Let $\mathcal{S}$ and $\mathcal{M}$ be an infinite state distributed system and the infinite Markov chain corresponding to $\mathcal{S}$, respectively. Let $\mathcal{C}_1, \mathcal{C}_2, \ldots$ be the strongly connected components of $\mathcal{S}$, and assume that there is a constant c such that $|\mathcal{C}_j| \leq c$ for all $j = 1, 2, \ldots$. Let $\mathcal{SP}$ be any specification for $\mathcal{S}$. Then $\mathcal{S}$ is deterministically weak stabilizing for $\mathcal{SP}$ under the strongly fair scheduler, if and only if it is probabilistically self-stabilizing for $\mathcal{SP}$ under the randomized scheduler.*

Proof. We only need to show that if $\mathcal{S}$ holds the possible convergence property for $\mathcal{SP}$ under the strong fair scheduler, then it holds the probabilistic convergence property for $\mathcal{SP}$ under the randomized scheduler.

Since each strongly connected component is of finite size, the component graph H of $\mathcal{S}$ is an infinite DAG. Let $v_i \in D$ be any sink of H. Since $\mathcal{S}$ holds the possible convergence property, there is a legitimate configuration in $\mathcal{C}_i$. Since $\mathcal{C}_i$ is a finite set, all the configurations in $\mathcal{C}_i$ are recurrent. Thus the evolution eventually reaches the legitimate configuration with probability 1, once it reaches $\mathcal{C}_i$.

We show that the evolution eventually reaches a legitimate configuration with probability 1, provided that it never reach a component $\mathcal{C}_i$ such that $v_i \in D$. Suppose that the evolution reaches a component $\mathcal{C}_i$ for some $v_i \notin D$. Since $\mathcal{C}_i$ is a finite set, it eventually leaves $\mathcal{C}_i$ and reaches another component $\mathcal{C}_j$ such that $(v_i, v_j) \in A$ with probability 1. For any fixed directed infinite path $X = v_{j_1}, v_{j_2}, \ldots$ in H, we show that the evolution visits a legitimate configuration with probability 1, provided that X is the path that the evolution traverses.

Let J be the set of indices j_t such that $\mathcal{C}_{j_t}$ contains a legitimate configuration. Since $\mathcal{S}$ holds the possible convergence property, J is an infinite set.

Consider any $j \in J$. Let $\gamma, \gamma' \in \mathcal{C}_j$ be any two configurations such that $\gamma \mapsto \gamma'$. Recall that the transition probability $p_{\gamma\gamma'}$ from γ to γ' is $2^{-|U|}$, where U is the set of enabled processes in γ, and thus $p_{\gamma\gamma'} \geq 2^{-N}$. Let $\lambda, \lambda' \in \mathcal{C}_j$ be any two configurations, which may not be adjacent. Since $\mathcal{C}_j$ is strongly connected, there is a directed path from λ to λ' and its length is less than c. Thus the probability that the evolution reaches λ' from λ is bounded from below by a constant $\epsilon = (2^{-N})^{c-1} > 0$.

Consider the following stochastic process: There are states $0, 1, \ldots$ States $1, 2, \ldots$ correspond to $v_{j_1}, v_{j_2}, \ldots$ State 0 represents the fact that the evolution visits a legitimate configuration. There is an edge from t to $t+1$ for any $t = 1, 2, \ldots$ If $j_t \notin J$, we assign the transition probability 1 to edge $(t, t+1)$. If $j_t \in J$, then we newly create an edge $(t, 0)$ and assign the transition probability ϵ to this edge. We also assign the transition probability $1 - \epsilon$ to edge $(t, t+1)$.

By the analysis of Success Runs in [21, Example 1.1], since $\sum_{j_t \in J} \epsilon = \infty$, the probability that the evolution starting in state 1, *i.e.*, a configuration in $\mathcal{C}_{j_1}$, eventually reaches state 0, *i.e.*, a legitimate configuration, is 1. □

For any strongly connected component $\mathcal{C}_i$, let $\mathcal{S}_i$ be the induced subgraph of $\mathcal{S}$ induced by $\mathcal{C}_i$. The Markov chain $\mathcal{M}_i$ corresponding to $\mathcal{S}_i$ is a one defined over $\mathcal{C}_i$ associated with the following transition probability matrix $\mathbf{P}_i$: For any edge (γ, γ') in $\mathcal{S}_i$, the transition probability from γ to γ' is $1/\delta^+_\gamma(\mathcal{S}_i)$. The next theorem is an extension of both Theorem 5 and Proposition 4.

Theorem 8 *Let $\mathcal{S}$ and $\mathcal{M}$ be an infinite state distributed system and the infinite Markov chain corresponding to $\mathcal{S}$, respectively. Let $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k$ and $H = (\{v_1, v_2, \ldots, v_k\}, A)$ be the strongly connected components of $\mathcal{S}$ and its component digraph, respectively,* i.e.*, the number of strongly connected components of $\mathcal{S}$ is finite. Assume that the Markov chain $\mathcal{M}_i$ corresponding to $\mathcal{S}_i$ is recurrent for all $i = 1, 2, \ldots, k$. Let $\mathcal{SP}$ be any specification for $\mathcal{S}$. Then $\mathcal{S}$ is deterministically weak stabilizing for $\mathcal{SP}$ under the strongly fair scheduler, if and only if it is probabilistically self-stabilizing for $\mathcal{SP}$ under the randomized scheduler.*

Proof. We only need to show that if $\mathcal{S}$ holds the possible convergence property for $\mathcal{SP}$ under the strong fair scheduler, then it holds the probabilistic convergence property for $\mathcal{SP}$ under the randomized scheduler.

Since $\mathcal{S}$ holds the possible convergence property for $\mathcal{SP}$, if $v_i \in D$, $\mathcal{C}_i$ must contain a legitimate configuration and the evolution eventually reaches a legitimate configuration with probability 1, once it reaches $\mathcal{C}_i$, since $\mathcal{M}_i$ is recurrent.

Suppose that the evolution does not reach a component $\mathcal{M}_i$ such that $v_i \in D$. Then it resides in a component $\mathcal{C}_j$ forever. Since $v_j \notin D$, there are configurations $\gamma \in \mathcal{C}_j$ and $\gamma' \in \mathcal{C}_\ell$ $(j \neq \ell)$ such that $(\gamma, \gamma') \in A$. Since $\mathcal{M}_j$ is recurrent, the evolution visits γ infinitely many times but transition $\gamma \mapsto \gamma'$ never occur, a contradiction. □

5 Discussion: Evaluating Time Complexity

The generally accepted metric for evaluating the theoretical time performance of (deterministic) self-stabilizing protocols is the *stabilization time*, that is, the maximum time before reaching a legitimate configuration. When the scheduling assumptions are arbitrary (that is, the proper scheduler can be tolerated), the stabilization time is simply the longest chain of consecutive illegitimate configurations. When fairness assumptions are necessary to guarantee convergence, stabilization time is described in terms of rounds. A round is recursively defined as the smallest portion of an execution where any process that is enabled in the beginning configuration either is scheduled for execution or becomes disabled (due to actions performed by other processes). A round may nevertheless comprise an arbitrarily high number of configurations. A downside of worst case analysis is that the timing performance that is obtained may not be representative of the actual performance of a given self-stabilizing algorithm.

Expected stabilization time was first investigated in the context of probabilistic stabilization, and a popular tool for obtaining expected stabilization time is the scheduler-luck game technique [23]. An execution is seen as a two-players game (algorithm and scheduler): the scheduler selects processes such that stabilization is delayed, while the algorithm executes random choices to force its luck and defeat the scheduler. Yet, worst case scheduler analysis often yields exponential expected stabilization time. Weakened forms of schedulers (where the scheduler choices are characterized by a probabilistic distribution) have been investigated [24], with contrasted results (the scheduler probabilistic distribution may yield infinite stabilization time).

A recent approach that permits to actually compute the expected stabilization time for *all* executions is due to Fallahi *et al.* [25]. The main underlying idea is to consider a probability distribution over individual transitions between configurations (hence defining an adaptive probability distribution for the scheduler), which permits to use Markov chains to evaluate the expected mean value of the time to reach a legitimate configuration starting from any illegitimate configuration. This approach has been implemented using a probabilistic model checker to obtain expected stabilization time for numerous examples, and is general enough to be used for self, probabilistic, and weak stabilizing protocols. For example, if a weak-stabilizing system satisfies the conditions given in Section 4, its expected stabilization time is finite and can be computed with their approach.

The follow-up paper [26] is dedicated to weak stabilization. In this paper, Fallahi and Bonakdarpour refine the general approach given in [25] and apply it to weak-stabilizing algorithm. Then, they introduce a second metric, that is built from a coarse-grained analysis of the structure of the state transition system. This graph-theoretic metric is based on the identification of the strongly connected components and their betweenness centrality. (The betweenness centrality quantifies the number of times a vertex acts as a bridge

along the shortest path between two other vertices.) The metric also incorporates the size and the density of the state transition system. It is noteworthy that their benchmarking example for illustrating their approach is the leader election algorithm we present in Subsection 3.2.

6 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 algorithms 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 finite state deterministic weak stabilizing algorithms can be automatically turned into the corresponding probabilistic self-stabilizing ones, provided that the scheduler is randomized. Although not all infinite state algorithms have this useful property, we show that some sufficient conditions for an infinite state algorithm to have this property. 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.

An obvious open problem is to find a weaker sufficient condition. Let $\mathcal{C}_1, \mathcal{C}_2, \ldots$ be the strongly connected components of a distributed system $\mathcal{S}$. If there is an i such that $\mathcal{M}_i$ is an infinite Markov chain and a configuration in $\mathcal{C}_i$ in $\mathcal{M}_i$ is transient, there is a specification $\mathcal{SP}$ such that $\mathcal{S}$ is deterministically weakly stabilizing for $\mathcal{SP}$ under the strongly fair scheduler, but it is not probabilistically self-stabilizing for $\mathcal{SP}$ under the randomized scheduler. The sufficient condition of Theorem 8 is thus best possible in this sense, when the number of strongly connected components is finite. On the other hand, when the number of strongly connected components is infinite, the sufficient condition of Theorem 7 may be slightly weakened.

A decision procedure to determine whether or not a given deterministically weak stabilizing algorithm for a given specification $\mathcal{SP}$ under the strongly fair scheduler is probabilistically self-stabilizing for $\mathcal{SP}$ under the randomized scheduler is also an interesting open problem.

References

[1] Stéphane Devismes, Sébastien Tixeuil, and Masafumi Yamashita. Weak vs. self vs. probabilistic stabilization. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, pages 681–688, Beijin, China, June 2008.

[2] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[3] Shlomi. Dolev. *Self-stabilization*. MIT Press, March 2000.

[4] Sébastien Tixeuil. *Algorithms and Theory of Computation Handbook, Second Edition*, chapter Self-stabilizing Algorithms, pages 26.1–26.45. CRC Press, Taylor and Francis Group, November 2009.

[5] Amos Israeli and Marc Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, pages 119–131, 1990.

[6] Maria Gradinariu and Sébastien Tixeuil. Conflict managers for self-stabilization without fairness assumption. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS 2007)*, page 46. IEEE, June 2007.

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

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

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

[10] Joffroy Beauquier, Christophe Genolini, and Shay Kutten. k-stabilization of reactive tasks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC 1998)*, page 318, 1998.

[11] 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.

[12] Ajoy Kumar Datta, Lawrence L. Larmore, Stéphane Devismes, and Sébastien Tixeuil. Fast leader (full) recovery despite dynamic faults. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 716–725, Cambridge, MA, USA, May 20-24 2013. IEEE.

[13] Mohamed G. Gouda. The theory of weak stabilization. In Ajoy Kumar Datta and Ted Herman, editors, *WSS*, volume 2194 of *Lecture Notes in Computer Science*, pages 114–123. Springer, 2001.

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

[15] Swan Dubois and Sébastien Tixeuil. A taxonomy of daemons in self-stabilization. Technical Report 1110.0334, ArXiv eprint, October 2011.

[16] Christian Boulinier, Franck Petit, and Vincent Villain. Synchronous vs. asynchronous unison. *Algorithmica*, 51(1):61–80, 2008.

[17] Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory requirements for silent stabilization. *Acta Inf.*, 36(6):447–462, 1999.

[18] 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, January 2007.

[19] Masafumi Yamashita and Tsunehiko Kameda. Computing on anonymous networks: Part i-characterizing the solvable cases. *IEEE Trans. Parallel Distrib. Syst.*, 7(1):69–89, 1996.

[20] 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.

[21] Pierre Brémaud. *Markov Chains – Gibbs Fields, Monte Carlo Simulation, and Queues*, volume 31 of *Texts in Applied Mathematics*. Springer Berlin / Heidelberg, 1999.

[22] Patrick Billingsley. *Probability and Measure*. Wiley Series in Probability and Mathematical Statistics, John Wiley & Sons, Inc., New York, 1995.

[23] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Analyzing expected time by scheduler-luck games. *IEEE Trans. Software Eng.*, 21(5):429–439, 1995.

[24] Yukiko Yamauchi, Sébastien Tixeuil, Shuji Kijima, and Masafumi Yamashita. Brief announcement: Probabilistic stabilization under probabilistic schedulers. In Marcos K. Aguilera, editor, *Distributed Computing - 26th International Symposium, DISC 2012, Salvador, Brazil, October 16-18, 2012. Proceedings*, volume 7611 of *Lecture Notes in Computer Science*, pages 413–414. Springer, 2012.

[25] Narges Fallahi, Borzoo Bonakdarpour, and Sébastien Tixeuil. Rigorous performance evaluation of self-stabilization using probabilistic model checking. In *Proceedings of the International Conference on Reliable Distributed Systems (SRDS 2013)*, pages 153–162, Braga, Portugal, September 2013. IEEE, IEEE Computer Society.

[26] Narges Fallahi and Borzoo Bonakdarpour. How good is weak-stabilization? In Teruo Higashino, Yoshiaki Katayama, Toshimitsu Masuzawa, Maria Potop-Butucaru, and Masafumi Yamashita, editors, *Stabilization, Safety, and Security of Distributed Systems - 15th International Symposium, SSS 2013*, volume 8255 of *Lecture Notes in Computer Science*, pages 148–162, Osaka, Japan, November 13-16 2013. Springer.