On Probabilistic Snap-Stabilization

Karine Altisen and Stéphane Devismes

VERIMAG UMR 5104, Université de Grenoble
{Karine.Altisen,Stephane.Devismes}@imag.fr
http://www-verimag.imag.fr

Abstract. In this paper, we introduce *probabilistic snap-stabilization*. We relax the definition of deterministic snap-stabilization without compromising its safety guarantees. In an unsafe environment, a probabilistically snap-stabilizing algorithm satisfies its safety property *immediately* after the last fault; whereas its liveness property is only ensured with probability 1.

We show that probabilistic snap-stabilization is more expressive than its deterministic counterpart. Indeed, we propose two probabilistic snap-stabilizing algorithms for a problem having no deterministic snap- or self-stabilizing solution: *guaranteed service leader election* in arbitrary anonymous networks. This problem consists in computing a correct answer to each process that initiates the question "Am I the leader of the network?", *i.e.*, (1) processes always computed the same answer to that question and (2) exactly one process computes the answer $true$.

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

Keywords: Snap-stabilization, probabilistic algorithms, leader election.

1 Introduction

Self-stabilization [14] is a versatile technique to withstand *any* transient fault in a distributed system: a self-stabilizing algorithm is able to recover, *i.e.,* reach a legitimate configuration, in finite time, regardless the *arbitrary* initial configuration of the system, and therefore also after the occurrence of transient faults. Thus, self-stabilization makes no hypotheses on the nature or extent of transient faults that could hit the system, and recovers from the effects of those faults in a unified manner. Such versatility comes at a price. After transient faults, there is a finite period of time, called the *stabilization phase*, before the system returns to a legitimate configuration. During this phase, there is no safety guarantee at all. In addition, a process cannot locally detect whether the system is actually in a legitimate configuration. Moreover, self-stabilizing algorithms may require a large amount of resources, *e.g.*, extra memory is usually required to crosscheck inconsistencies. Finally, symmetries occurring in the initial configuration could cause a problem to be impossible to solve, *e.g.*, leader election [27] and token passing [21] have no deterministic self-stabilizing solutions in anonymous networks. To cope with those

issues, two categories of variants of self-stabilization have been introduced: *weakened* and *strengthened* forms of self-stabilization.

Related Work. *Weakened forms* of self-stabilization have been introduced to cope with impossibility results, reduce the stabilization time, or limit the resource consumption. *Weak stabilization* [20] stipulates that starting from *any* initial configuration, *there exists* a run that eventually reaches a legitimate configuration. Unlike for self-stabilization, token passing and leader election have weak stabilizing solutions in anonymous networks [13]. *k-stabilization* [3] prohibits some of the configurations from being initial, as an initial configuration may only be the result of at most k faults. There are k-stabilizing token passing algorithms that guarantee small convergence time depending only on k [3]. *Probabilistic self-stabilization* [22] weakens the convergence property: starting from any initial configuration, the system converges to a legitimate configuration with probability 1. Problems such as token passing and leader election in anonymous networks have probabilistic self-stabilizing solutions [22, 15].

Strengthened forms of self-stabilization have been mainly introduced to offer stringent safety guarantees. A *fault containing* self-stabilizing algorithm [19] ensures that when few faults hits the system, the faults are both spatially and temporally contained. "Spatially" means that if only few faults occur, those faults cannot be propagated further than a preset radius around the corrupted nodes. "Temporally" means quick stabilization when few faults occur. A *superstabilizing algorithm* [16] is self-stabilizing and has two additional properties. In presence of single topological change, it recovers fast, and a safety predicate, called a *passage* predicate, should be satisfied along the stabilization. Finally, *(deterministic) snap-stabilization* [8] offers strong safety guarantees: regardless of the configuration to which transient failures drive the system, after the failures stop, a snap-stabilizing system immediately resumes correct behavior. Precisely, a snap-stabilizing algorithm guarantees that any computation started after the faults will operate correctly. However, we have no guarantees for those executed all or a part during faults. Actually, snap-stabilization is often used to offer *user-centric* guarantees: the problems considered consist of executing finite tasks called *services*: a service is started by some initiating process and terminates by providing a result to that initiator. The goal is to ensure that, starting from any configuration, a service eventually starts if requested by some process; and every started service is computed correctly. We call those problems *guaranteed service problems*.

Contribution. We introduce a new property called *probabilistic snap-stabilization*, a probabilistic variant of (deterministic) snap-stabilization. Just as for the probabilistic extension of self-stabilization, we choose to adopt a "Las Vegas" approach and relax the definition of snap-stabilization without altering its safety guarantees. Considering a specification as the conjunction of safety and liveness properties, a probabilistically snap-stabilizing algorithm immediately satisfies the safety property at the end of the faults, whereas the liveness property is ensured with probability 1.

We show that probabilistic snap-stabilization is strictly more expressive than its deterministic counterpart, as we give two probabilistic snap-stabilizing algorithms for a problem having no deterministic self- or snap-stabilizing solution: *guaranteed service leader election* in anonymous networks. This problem consists in computing a correct answer to each process that initiates the question "Am I the leader of the net-

work ?", *i.e.*, (1) processes always computed the same answer to that question and (2) exactly one process computes the answer $true$. Our solutions being probabilistically snap-stabilizing, the answers are delivered within an almost surely finite time; however, any delivered answer is correct, regardless the arbitrary initial configuration and provided that the question has been properly started.

Our two algorithms work in the locally shared memory model. The first solution, $\mathcal{S}_{\mathcal{GSLE}}$, assumes a synchronous daemon. The second, $\mathcal{A}_{\mathcal{GSLE}}$, assumes an unfair (distributed) daemon, the most general daemon of the model. Both algorithms need an additional assumption:[1] the knowledge of a bound B such that $B < n \leq 2B$, where n is the number of processes. The memory requirement of both algorithms is in $O(\log n)$ bits per process. The expected delay, response, and service times of $\mathcal{S}_{\mathcal{GSLE}}$ are each $O(n)$ rounds, while these times are $O(n^2)$ rounds for $\mathcal{A}_{\mathcal{GSLE}}$. If we add the assumption that processes know an upper bound, D, on the diameter of the network, the expected time complexity of $\mathcal{A}_{\mathcal{GSLE}}$ can be made $O(D.n)$ rounds.

Roadmap. In the next section we define the computational model. In Section 3, we introduce *probabilistic snap-stabilization*. In the same section, we formally define the guaranteed service problem, and give one example, namely guaranteed service leader election. In Section 4, we propose our two probabilistic snap-stabilizing algorithms. We conclude in Section 5.

2 Preliminaries

Below, we define a general model to handle probabilistic algorithms in anonymous networks. It settles the formal context in which probabilistic snap-stabilization is defined.

Distributed Systems. We consider distributed systems of $n \geq 2$ *anonymous* processes, *i.e.*, they all have the same program, and have no parameter (such as an identity) permitting them to be differentiated. Each process p can directly communicate with a subset of other processes, called its *neighbors*. Communication is *bidirectional*. We model a distributed system as a simple undirected connected graph $G = (V, E)$, where V is the set of processes and E is a set of edges representing (direct) communication relations. Every processor p can distinguish its neighbors using a local labeling. All labels of p's neighbors are stored into the set $\mathcal{N}_p$. Moreover, we assume that each process p can identify its local label in the set $\mathcal{N}_q$ of each neighbor q. By an abuse of notation, we use p to designate both the process p itself, and also its local labels.

Computational Model. We assume the *locally shared memory model* [14], where each process communicates with its neighbors using a finite set of locally shared registers, henceforth called simply *variables*. A process can read its own variables and those of its neighbors, but can write only to its own variables. Each process operates according to its *program*. A *(distributed) algorithm* $\mathcal{A}$ is defined to be a collection of n *programs*, each operating on a single process. The program of each process consists of a finite set of actions of the form: $\langle guard \rangle \longrightarrow \langle statement \rangle$. The *guard* of an action in the program of a process p is a Boolean expression involving the variables of p and its neighbors; the *statement* updates some variables of p. An action can be executed only

[1] We otherwise prove that our problem is probabilistically unsolvable.

if its guard evaluates to *true*. The *state* of a process p is the vector consisting of the values of all its variables. The set of all variables of p is noted $\texttt{Var}_p$, and for a variable $v \in \texttt{Var}_p$, $\texttt{Dom}_p(v)$ denotes the set of all possible values of v. A variable v in $\texttt{Var}_p$ can be either a *discrete* variable, which ranges over a discrete (but possibly infinite) set of values (namely, $\texttt{Dom}_p(v)$ is a countable set) or v is a real variable for which $\texttt{Dom}_p(v)$ is a given interval of reals.[2] The set S_p of all possible states of process p, is defined to be the Cartesian product of the sets $\texttt{Dom}_p(v)$, over all $v \in \texttt{Var}_p$.

A *configuration*, γ, consists of one state, $\gamma(p) \in S_p$ for each process p, *i.e.*, $\gamma \in \mathcal{C} \stackrel{\texttt{def}}{=} \prod_{p \in V} S_p$, the set of all possible configurations. We denote by $\gamma(p).x$ the value of the variable x of process p in γ.

An action in the program of a process p is said to be *enabled in* γ if and only if its guard is true in γ, and we say p is enabled in γ if and only if some action in the program of p is enabled in γ. Let $Enabled(\gamma)$ be the set of processes which are enabled in γ.

When the system is in configuration γ, a *daemon* selects a subset ϕ of $Enabled(\gamma)$ to execute an *(atomic) step*: every process of ϕ atomically executes one of its enabled actions, leading to a configuration γ'. So, a daemon is a function $d : \mathcal{C}^+ \to 2^V$: Considering the finite sequence of configurations $\rho = (\gamma_0, ..., \gamma_k)$ through which the system has evolved, the daemon gives the next chosen subset $d(\rho) \subseteq Enabled(\gamma_k)$ of enabled processes that will execute in the next step. Let $\mathcal{D}_{all}$ be the set of all possible daemons.

Deterministic Algorithms. We model a distributed *deterministic* algorithm $\mathcal{A}$ as a function $F_{\mathcal{A}} : \mathcal{C} \times 2^V \to \mathcal{C}$, *i.e.*, the next configuration is only determined by the current configuration and the set of processes that will execute the next step. A *run* of a deterministic algorithm $\mathcal{A}$ under the daemon d is an infinite sequence $r = (\gamma_i)_{i \geq 0} \in \mathcal{C}^\omega$ inductively defined as follows: $\gamma_0 \in \mathcal{C}$ and $\forall i \geq 0$, $\gamma_{i+1} = F_{\mathcal{A}}(\gamma_i, d(\gamma_0, \ldots, \gamma_i))$, *i.e.*, γ_{i+1} is obtained from γ_i by an atomic step of all processes in $d(\gamma_0, \ldots, \gamma_i)$.[3]

Probabilistic Algorithms. The modeling of probabilistic distributed algorithms is a bit more intricate since it has to handle the interactions between probabilities and non-determinism, like in Markov Decision Process [25]. In our case, probabilities come from the use of random functions in process programs, while non-determinism is due to asynchrony. Below, we provide a concise model and semantics using infinite Markov chains; alternative models can be found in [4, 17].

Following the literature [18], for every process p and for every variable $v \in \texttt{Var}_p$, we define $\sigma_{p,v}$, the *sigma-field* (informally, the set of possible events on the value of v) associated with v:

- If v is discrete, then $\sigma_{p,v}$ is the *discrete sigma-field* on $\texttt{Dom}_p(v)$ formed by the power-set of $\texttt{Dom}_p(v)$.
- If v is real, then $\sigma_{p,v}$ is the *Borel sigma-field* of $\texttt{Dom}_p(v)$.

The *sigma-field over* $\mathcal{C}$, denoted by $\Sigma_{\mathcal{C}}$, is the *minimal sigma-field* of $\mathcal{C}$, generated by the subsets of the form $\prod_{p \in V} \prod_{v \in \texttt{Var}_p} s_{p,v}$, where $s_{p,v} \in \sigma_{p,v}$.

We model a distributed *probabilistic* algorithm $\mathcal{A}$ as a family of probability kernels $\mu_d : \mathcal{C}^+ \times \Sigma_{\mathcal{C}} \to [0, 1]$ indexed on any possible daemon $d \in \mathcal{D}_{all}$. Namely, for any fixed $s \in \Sigma_{\mathcal{C}}$, the function $x \to \mu_d(x, s)$ is $\Sigma_{\mathcal{C}^+}$-measurable, and for any prefix $p \in \mathcal{C}^+$,

[2] Our algorithms only deal with finite-state variables, whereas our model is more general.

[3] Note that $F_{\mathcal{A}}(\gamma_i, \emptyset) = \gamma_i$.

the function $x \rightarrow \mu_d(p, x)$ is a probability measure. Considering the finite sequence of configurations $\rho = (\gamma_0, ..., \gamma_k)$ through which the system has evolved using the daemon d, $x \rightarrow \mu_d(\rho, x)$ describes the probability law for the next configuration. In particular, $\mu_d(\rho, \{\gamma_{k+1}\})$ is the probability of reaching configuration γ_{k+1} after prefix ρ. The sequence $r = (\gamma_i)_{i \geq 0} \in \mathcal{C}^\omega$ is a *run* of $\mathcal{A}$ under the daemon d if and only if there exists a sequence of choices of d, $(\phi_i)_{i \geq 0} \in (2^V)^\omega$, such that $\forall i \geq 0$, $\phi_i = d(\gamma_0, ..., \gamma_i)$ and there exists $s \in \Sigma_\mathcal{C}$ such that $\gamma_{i+1} \in s$ and $\mu_d(\gamma_0, ..., \gamma_i, s) > 0$.

Given a daemon d and a probabilistic algorithm $\mathcal{A}$, let R_n be a random variable over $\mathcal{C}^n$, representing a prefix of size $n \in \mathbb{N}$ of a run: $R_n = \Gamma_0 \cdots \Gamma_n$, where each Γ_{i+1} is obtained from Γ_i after one more step of $\mathcal{A}$. Note that the sequence $\left(R_n\right)_{n \in \mathbb{N}}$ is a Markov chain, since the possible values for R_{n+1} on $R_0, \ldots, R_{n-1}, R_n$ only depends on R_n. The probability space for the model is naturally derived from the usual Markov chain theory. Let R be a random variable on $\mathcal{C}^\omega$ and $\mathcal{R} \subseteq \mathcal{C}^\omega$ be a measurable set of runs of $\mathcal{A}$. Let $\gamma_0 \in \mathcal{C}$. We denote by $\mathsf{Pr}_\mathcal{A}^{d,\gamma_0}(\mathcal{R})$ the probability that a run of $\mathcal{R}$ occurs on $R_0 = \gamma_0$.[4]

Daemon Families. We only consider here daemons that are *proper*, namely for every run $(\gamma_i)_{i \geq 0}$ of a distributed algorithm $\mathcal{A}$, under a proper daemon d, for every $i \geq 0$, $Enabled(\gamma_i) \neq \emptyset \Rightarrow d(\gamma_0, \ldots, \gamma_i) \neq \emptyset$. Daemons are usually classified into *families* according to their fairness property. The family of *unfair (distributed) daemons*, noted $\mathcal{D}_U$, is the set of all proper daemons. A daemon d is said to be *synchronous* if and only if for every run $(\gamma_i)_{i \geq 0}$ under d, for every $i \geq 0$, $d(\gamma_0, \ldots, \gamma_i) = Enabled(\gamma_i)$. We denote by $\mathcal{D}_S$ the set of all synchronous daemons. We denote by $\mathcal{R}_\mathcal{A}^\mathcal{D}$ the set of all possible runs of $\mathcal{A}$ using a daemon of family $\mathcal{D}$. We denote by $\mathcal{R}_\mathcal{A}^{\mathcal{D},\gamma_0}$ the subset of runs of $\mathcal{R}_\mathcal{A}^\mathcal{D}$ starting from configuration γ_0.

Rounds. In some run $r = (\gamma_i)_{i \geq 0}$ of an algorithm $\mathcal{A}$ under a daemon d, we say that a process p is *neutralized* in the step γ_i, γ_{i+1} if p is enabled in γ_i and not enabled in γ_{i+1}, but does not execute any action between these two configurations.

We use the notion of *round* to evaluate the time complexity of $\mathcal{A}$. This notion captures the execution rate of the slowest processor in every run. The first *round* of r, noted r', is the minimal prefix of r in which every process that are enabled in γ_0 either execute an action or become neutralized. Let r'' be the suffix of r starting from the last configuration of r'. The second round of r is the first round of r'', and so forth. Notice that, by definition, under a synchronous daemon, each round lasts exactly one step.

3 Snap-Stabilization and Guaranteed Service Problems

In this section, we first recall the definition of deterministic snap-stabilization. Then, we introduce the probabilistic snap-stabilization. Finally, we define the guaranteed service problems, and we instantiate this notion to define guaranteed service leader election.

Following [1], we express the *specification* SP of an algorithm as a set of runs, or equivalently as a predicate that any run should satisfy: $SP \subseteq \mathcal{C}^\omega$. We use the usual result [1, 23] that a specification can always be expressed as the conjunction of a safety property and a liveness property: $SP = Safe \cap Live$.

[4] We fix γ_0, since we do not assume any distribution on the initial configurations.

In (self- or snap-) stabilization, we consider the system right after the occurrence of the last fault, *i.e.*, we study the system starting from an arbitrary configuration reached due to the occurrence of transient faults, but from which no additional fault will ever occur. By abuse of language, this *arbitrary* configuration is referred to as *initial* configuration of the system. Deterministic snap-stabilization has been defined as follows:

Definition 1 (Deterministic Snap-Stabilization [8]) *An algorithm $\mathcal{A}$ is* snap-stabilizing *w.r.t. a specification $SP = Safe \cap Live$ and a family of daemons $\mathcal{D}$ iff(def) $\forall r \in \mathcal{R}_{\mathcal{A}}^{\mathcal{D}}$, $r \in SP$ (*i.e.*, $r \in Safe$ and $r \in Live$).*

The idea behind *probabilistic snap-stabilization* is to weaken deterministic snap-stabilization without compromising its strong safety guarantees. The safety part remains unchanged, but, we allow the algorithm to compute for a possibly long, yet almost surely finite, time.

Definition 2 (Probabilistic Snap-Stabilization) *An algorithm $\mathcal{A}$ is* probabilistically snap-stabilizing *w.r.t. a specification $SP = Safe \cap Live$ and a family of daemons $\mathcal{D}$ iff(def)*
Strong Safety: $\forall r \in \mathcal{R}_{\mathcal{A}}^{\mathcal{D}}$, $r \in Safe$, *and*
Almost Surely Liveness: $\forall \gamma_0 \in \mathcal{C}, \forall d \in \mathcal{D}, \mathsf{Pr}_{\mathcal{A}}^{d,\gamma_0}(Live) = 1$.

Almost all snap-stabilizing solutions proposed so far solve *guaranteed service* problems, *i.e.*, problems consisting in executing finite tasks upon the request at some process. Propagation of Information with Feedback [10] is an example of such a problem: when an application at a given process p needs to broadcast some data, the service consists in ensuring that all processes eventually acknowledge the receipt of the data and that p eventually receives acknowledgment from all processes. Generally, a guaranteed service problem consists in ensuring three properties:

1. If an application at some process continuously requires the execution of the service, then the process — called *the initiator* — eventually starts a computation of the service (involving the whole or a part of the network).
2. Every started service eventually ends by a decision at its initiator, allowing it to get back a *result*.
3. Every result obtained from any started service is correct *w.r.t.* the service.

To formalize these properties, we use the following predicates, where $p \in V$, $s, s' \in S_p$, and $\gamma_0 \dots \gamma_t \in \mathcal{C}^+$:

- *Request*(s) means that the state s indicates to p that some application needs an execution of the service.
- *Start*(s, s') means that p starts a computation of the service by switching its state from s to s'.
- *Result*(s, s') means that p executes the decision event to get back the result of a computation by switching its state from s to s'.
- *Correct-Result*$(\gamma_0 \dots \gamma_t, p)$ means that the computed result is correct *w.r.t.* the service.

Definition 3 (Specification for Guaranteed Service) *A* guaranteed service specification *is a specification $\mathcal{S}_{gs} = Safe_{gs} \cap Live_{gs}$ where $Safe_{gs}$ and $Live_{gs}$ are defined as follows: let $r = (\gamma_i)_{i \geq 0}$,*

(a) $r \in Safe_{gs}$ *if and only if* $\forall k \geq 0, \forall p \in V, \big(\textit{Result}(\gamma_k(p), \gamma_{k+1}(p)) \wedge \exists l < k, \textit{Start}(\gamma_l(p), \gamma_{l+1}(p)) \Rightarrow \textit{Correct-Result}(\gamma_0 \dots \gamma_k, p)\big)$.
(b) $r \in Live_{gs}$ *if and only if the following two conditions hold:*
(1) $\forall k \geq 0, \forall p \in V, \exists l \geq k, \big(\textit{Request}(\gamma_l(p)) \Rightarrow \textit{Start}(\gamma_l(p), \gamma_{l+1}(p))\big)$.
(2) $\forall k \geq 0, \forall p \in V, \big(\textit{Start}(\gamma_k(p), \gamma_{k+1}(p)) \Rightarrow (\exists l > k, \textit{Result}(\gamma_l(p), \gamma_{l+1}(p)))\big)$.

The safety condition *(a)* means that when a result is delivered, it is *correct*, provided that the task that computed the result was started. The liveness condition *(b)* means that *(1)* it cannot happen that an application continuously requests the service without being served and *(2)* any started computation eventually delivers a result.

When a guaranteed service solution is snap-stabilizing, this implies that whatever the initial configuration is, every started service delivers within a finite time a correct result to its initiator. Considering probabilistic snap-stabilization, when a result is delivered, it is *correct*, provided that the service corresponding to this result has been properly started (similar to the deterministic case), but the time elapsed between the starting and the corresponding result is only *almost surely finite*.

The classical definition of *leader election* requires that there should be always at most one leader, and if a process is designated as leader it remains leader forever (safety), and that eventually there should exist a leader (liveness). This specification can be turned into a guaranteed service specification. Every process may initiate a computation (*i.e.*, a question) to know whether it is the leader. The results should be consistent, *i.e*, the result from any initiated leader-computation by some process p is a *constant* truth value, and exactly one process always obtains the result $true$ to its initiated leader-computations.

In the following definition, we use the predicate $Unique(\gamma, p)$, which is true if and only if there is exactly one process p that designates itself as *leader* in configuration γ.

Definition 4 (Guaranteed Service Leader Election Specification) *The* guaranteed service leader election specification $\mathcal{S}_{gs,leader}$ *is given using Definition 3 of guaranteed service, where the predicate Correct-Result is instantiated as follows.*

Correct-Result$_{leader}(\gamma_0 \dots \gamma_k, p)$ *is true if and only if* $\exists p^* \in V$ *such that:*

- $Unique(\gamma_k, p^*)$ *and*
- $\forall i \in \{i^*, ..., k\}, Unique(\gamma_i, p^*)$, *where* $i^* = \min\big\{i \in \{1, ..., k\} : \exists q \in V, \textit{Result}(\gamma_{i-1}(q), \gamma_i(q)) \wedge \exists l < i, \textit{Start}(\gamma_l(q), \gamma_{l+1}(q))\big\}$.

The first item of the definition ensures that there exists a unique leader at the completion of a previously started leader-computation. The second item ensures that all results obtained from started leader-computations are consistent, *i.e.,* the leader is the same.

4 Probabilistic Snap-Stabilizing Guaranteed Service Leader Election

We now deal with a problem having no deterministic solution: the *guaranteed service leader election in anonymous networks* [2]. We first propose a probabilistic snap-stabilizing solution, called $\mathcal{S}_{\mathcal{GSLE}}$ (*Synchronous Guaranteed Service Leader Election*),

which assumes a *synchronous* daemon. Then, we slightly modify the solution to obtain a probabilistically snap-stabilizing algorithm, called $\mathcal{A}_{\mathcal{GSLE}}$ (*Asynchronous Guaranteed Service Leader Election*), which works under an *unfair (distributed)* daemon.

First, note that, without an additional assumption on the system (*e.g.*, the knowledge of some upper bound on some network global parameter), there is no probabilistic guaranteed service leader election in our setting. We prove this claim by reducing the guaranteed service leader election to the ring-size counting problem, which has been proven to be probabilistically unsolvable in our setting [26]. So, following the ideas in [24], we assume that the processes know a bound B on n such that $B < n \leq 2B$.

4.1 Synchronous Settings

$\mathcal{S}_{\mathcal{GSLE}}$ is made of three modules — `Election`, `Pulse`, and `Service` — which run concurrently: at each local step, a process simultaneously executes an action of each module where it is enabled. `Election` is the main module. It uses the output of `Pulse` to perform probabilistic *self-stabilizing* leader election. In `Election`, each process p maintains a Boolean variable $p.\texttt{me}$, which states whether it is candidate for the election. The module executes in cycles. Each cycle computes a Boolean output `oneLdr` at each process, stating whether a unique leader exists; it is computed by testing whether there is a unique process p^* which is candidate in the configuration γ ending the cycle, *i.e.*, if the predicate $Unique(\gamma, p^*) \stackrel{\texttt{def}}{=} \gamma(p^*).\texttt{me} \wedge \forall p \in V, p \neq p^* \Rightarrow \neg\gamma(p).\texttt{me}$ holds.

If there is no leader at the end of the cycle, the next cycle attempts to elect a leader by modifying the variables `me`. Once a cycle succeeds in electing a leader, all processes compute the output $true$ in their variable `oneLdr` and during the remaining cycles their variable `me` stays constant: once elected, the leader remains stable.

Conversely, if `oneLdr` is $false$ at the end of a cycle, then every process p randomly and synchronously chooses a new value for $p.\texttt{me}$: p resets $p.\texttt{me}$ to v, where v is a random Boolean value which is $true$ with probability α_{vote}, respectively $false$ with probability $1 - \alpha_{vote}$. In other words, the new random Boolean value v of $p.\texttt{me}$ follows a Bernoulli distribution of constant parameter α_{vote}, noted $v \hookrightarrow \mathcal{B}(\alpha_{vote})$ in the sequel. Hence, if there is not a leader at the end of a cycle, then a leader is elected during the next cycle with positive probability.

Processes should be synchronized in order to obtain a consistent output at the end of a cycle. This is the aim of the second module, `Pulse`, which is actually a *self-stabilizing synchronous unison* algorithm. In synchronous systems, a self-stabilizing unison consists in implementing a logical `clock` at each process such that, once the system is in a legitimate configuration, all local clocks have the same value and increment at each step; we denote by P_{SU} the predicate defining all legitimate configurations of the synchronous unison. Once `Pulse` achieves its legitimacy, all processes execute cycles synchronously. The output `oneLdr` computed during each cycle started in a configuration satisfying P_{SU} will be both consistent (*i.e.*, all processes will have the same value for `oneLdr`), and correct (*i.e.*, `oneLdr` will be $true$ if and only if there is unique leader in the system).

The composition of the two aforementioned modules is only probabilistically self-stabilizing. So, to ensure the safety part of $\mathcal{S}_{gs,leader}$, we cannot let a process directly

evaluate its variable `me` to state whether it is the leader, even if `oneLdr` is currently $true$. To guarantee that, after starting a question, a process delivers a correct result, we use an additional module, `Service`. The aim of this module is to delay the processing of each question (service), so that the initiator is able to consult its variables `me` and `oneLdr` only when they are correctly evaluated. To that end, p should (at least) wait for the first cycle of `Election` started after the system has reached a configuration satisfying P_{SU}. After this waiting time, an initiator p can consult $p.\texttt{me}$ and $p.\texttt{oneLdr}$ at the end of each cycle; when $p.\texttt{oneLdr}$ is $true$, it delivers the result $p.\texttt{me}$. Below, we give more details about the three modules of $\mathcal{S}_{\mathcal{GSLE}}$.

Module `Pulse`. We consider a slightly updated version of the unison algorithm proposed in [7], which we call $\mathcal{U}$. Actually, we add a local observer into the program of $\mathcal{U}$. This observer will be used by Module `Service` to guarantee service.

$\mathcal{U}$ is a unison algorithm with a bounded number of clock values: the clock of every process takes a value in $\{-\kappa, ..., \xi - 1\}$; however the negative values are only used during the stabilization phase. Once $\mathcal{U}$ has stabilized, the values of the clocks increment using the function Φ from 0 to $\xi - 1$, then back to 0, and so forth. Consequently, $P_{SU} \stackrel{\texttt{def}}{=} \forall p, q \in V, p.\texttt{clock} = q.\texttt{clock} \wedge p.\texttt{clock} \geq 0$. In [5], $\mathcal{U}$ is shown to be self-stabilizing for the synchronous unison in any anonymous synchronous network if $\kappa \geq n - 2$ and $\xi \geq n + 1$. Here, processes do not know n, but they know a value B such that $n \leq 2B$. So, it is sufficient to take $\kappa = 2B - 2$ and $\xi \geq 2B + 1$, to guarantee the self-stabilization of $\mathcal{U}$ in any topology. Moreover, the stabilization time of $\mathcal{U}$, called ST, is shown in [5] to be less or equal to $n + \kappa + \mathcal{D}$ steps in synchronous settings, where $\mathcal{D}$ is the diameter of the network. So, $ST \leq 6B$ steps (or equivalently $6B$ rounds).

We add to $\mathcal{U}$ an observer in order to obtain a guaranteed service property: as explained before, we need a mechanism that allows any process p to locally decide whether the configuration satisfies P_{SU}. To achieve that, p needs only to wait at least $6B$ steps, because $ST \leq 6B$. So, we add a variable $p.\texttt{cnt}$, which takes a value in $\{0, ..., \texttt{CMax}\}$, where $\texttt{CMax} = 6B$. At each step, $p.\texttt{cnt}$ is decremented if it is positive. We also add two functions that can be called in Module `Service`: $p.\texttt{init}()$ resets $p.\texttt{cnt}$ to `CMax`; and $p.\texttt{OK}()$ returns $true$ if and only if $p.\texttt{cnt}$ equals 0. Note that the local observers do not disturb the stabilization of $\mathcal{U}$, because they only write to a dedicated additional variable. Furthermore, starting from any configuration, for every process p, if $p.\texttt{init}()$ is called, $p.\texttt{OK}()$ returns $true$ `CMax` steps (resp. rounds) later and p has the guarantee that the configuration satisfies P_{SU}.

Module `Election`. This module (Algorithm 1) executes in cycles. Processes use their clocks in order to synchronize. Once all processes are synchronized, each cycle is performed in three phases. Each phase lasts φ steps and consists of an initialization step followed by a computation that is propagated to all processes. So, φ should be at least strictly greater than the diameter of the network: we set φ to $2B$. To easily distinguish each phase, we can set the size of the cycle to any value ξ greater or equal to $3\varphi = 6B$, a cycle starts when clocks reset to 0, and every process can determine in which phase it is thanks to its `clock`. For time complexity issues, we set ξ to exactly $6B$ since this also meets the condition for $\mathcal{U}$, *i.e.*, $\xi = 6B \geq 2B + 1$.

Each process p starts the *first phase* of a cycle (Line 1) by resetting $p.\texttt{me}$ and $p.\texttt{parent}$. The new value of $p.\texttt{me}$ depends on the result of the previous cycle: if a

Algorithm 1 Module `Election`, for every process p

Input: $p.\texttt{clock} \in \{-\kappa, ..., \xi - 1\}$: variable from Module `Pulse`
Variables:
$p.\texttt{me}$, $p.\texttt{oneLdr}$: Boolean
$p.\texttt{parent} \in \mathcal{N}_p \cup \{\texttt{null}\}$
$p.\texttt{stSize} \in \{1, ..., 2B\}$
Macros:
$\texttt{cmpMe} = \textbf{if } p.\texttt{oneLdr} \textbf{ then } p.\texttt{me} \textbf{ else } v \hookrightarrow \mathcal{B}(\alpha_{vote})$
$\texttt{branches} = \{q \in \mathcal{N}_p : q.\texttt{me} \vee q.\texttt{parent} \neq \texttt{null}\}$
$\texttt{cmpPar} = \textbf{if } p.\texttt{me} \vee p.\texttt{parent} \neq \texttt{null} \vee \texttt{branches} = \emptyset \textbf{ then } p.\texttt{parent} \textbf{ else } q \in \texttt{branches}$
$\texttt{children} = \{q \in \mathcal{N}_p : q.\texttt{parent} = p\}$
Actions:

1: $p.\texttt{clock} = \xi - 1 \mapsto p.\texttt{me} \leftarrow \texttt{cmpMe}; p.\texttt{parent} \leftarrow \texttt{null}$
2: $p.\texttt{clock} \in \{0, ..., \varphi - 2\} \mapsto p.\texttt{parent} \leftarrow \texttt{cmpPar}$
3: $p.\texttt{clock} = \varphi - 1 \mapsto p.\texttt{stSize} \leftarrow 1$
4: $p.\texttt{clock} \in \{\varphi, ..., 2\varphi - 2\} \mapsto p.\texttt{stSize} \leftarrow 1 + \sum_{q \in \texttt{children}} q.\texttt{stSize}$
5: $p.\texttt{clock} = 2\varphi - 1 \mapsto p.\texttt{me} \leftarrow p.\texttt{me} \wedge p.\texttt{stSize} > B; p.\texttt{oneLdr} \leftarrow p.\texttt{me}$
6: $p.\texttt{clock} \in \{2\varphi, ..., 3\varphi - 2\} \mapsto p.\texttt{oneLdr} \leftarrow p.\texttt{oneLdr} \vee \bigvee_{q \in \mathcal{N}_p} q.\texttt{oneLdr}$

process p believes that there is no unique leader (*i.e.*, $p.\texttt{oneLdr} = false$), p randomly chooses a new value for $p.\texttt{me}$ to decide if it is candidate in the new phase; otherwise, $p.\texttt{me}$ remains unchanged. The remainder of the phase (Line 2) consists in building a forest, where a process r is a tree root if and only if $r.\texttt{me} = true$. In the *second phase* of the cycle (Lines 3 and 4), each candidate (*i.e.*, each process whose variable `me` is $true$) computes the number of nodes in its own tree. Finally, the initialization step of the *third phase* (Line 5) is crucial for the candidates, if any. For every candidate p, if p has more than B nodes in its tree, a majority of nodes are in its tree since $n \leq 2B$. In that case, p is the only candidate in that situation. Consequently p is the leader and it sets both $p.\texttt{oneLdr}$ and $p.\texttt{me}$ to $true$. In either case, both $p.\texttt{oneLdr}$ and $p.\texttt{me}$ are set to $false$. All non-candidate processes also set their variables `oneLdr` and `me` to $false$ in this initialization step. The remainder of the phase (Line 6) allow propagation of the result to all processes so that when the next cycle begins, every process satisfies $\texttt{oneLdr} = true$ if and only if a unique leader is elected.

Module `Service`. Module `Service` (Algorithm 2) achieves snap-stabilizing guaranteed service. When a process p is available ($p.\texttt{status} = \texttt{Out}$, *i.e.*, p is not currently processing a service) and when it is requested (*i.e.*, $\mathit{Request}(s) \stackrel{\text{def}}{=} s.\langle\texttt{need}\rangle()$), it starts the computation of the service by switching $p.\texttt{status}$ to `Wait` (*i.e.*, $\mathit{Start}(s, s') \stackrel{\text{def}}{=} s.\texttt{status} = \texttt{Out} \wedge s'.\texttt{status} = \texttt{Wait}$) and resetting the observation in `Pulse` (Line 7). Then, p waits until $p.\texttt{OK}() \wedge p.\texttt{clock} = \xi - 1$: this waiting phase ensures that p will only consider cycles fully executed after the stabilization of `Pulse`. When this period has elapsed, p switches $p.\texttt{status}$ from `Wait` to `In` (Line 8), meaning that it is now allowed to consider the outputs computed by Module `Election`, which are available only at cycle completions. So, p delivers a result ($p.\texttt{me}$) only when $p.\texttt{oneLdr}$ is $true$ at the end of such a cycle (Line 9); in this case p switches $p.\texttt{status}$ from `In` to `Out` to inform the application of the availability of the result (*i.e.*, $\mathit{Result}(s, s') \stackrel{\text{def}}{=} s.\texttt{status} =$

$\texttt{In} \wedge s'.\texttt{status} = \texttt{Out}$). Note that, from the application point of view, this result is guaranteed *provided that* its request has been properly handled and the corresponding service properly started (Line 7).

Algorithm 2 Module `Service`, for every process p

Inputs:

$p.\texttt{clock} \in \{-\kappa, ..., \xi - 1\}$	: variable from Module `Pulse`
$p.\texttt{me}$, $p.\texttt{oneLdr}$: Boolean	: variables from Module `Election`
$\texttt{init}()$, $\texttt{OK}()$	: functions from Module `Pulse`
$\langle\texttt{need}\rangle()$, $\langle\texttt{deliver}\rangle(Boolean)$	: functions from the application

Variable: $p.\texttt{status} \in \{\texttt{Out}, \texttt{Wait}, \texttt{In}\}$

Actions:

7: $p.\texttt{status} = \texttt{Out} \wedge p.\langle\texttt{need}\rangle() \mapsto p.\texttt{status} \leftarrow \texttt{Wait}; p.\texttt{init}()$

8: $p.\texttt{status} = \texttt{Wait} \wedge p.\texttt{OK}() \wedge p.\texttt{clock} = \xi - 1 \mapsto p.\texttt{status} \leftarrow \texttt{In}$

9: $p.\texttt{status} = \texttt{In} \wedge p.\texttt{clock} = 3\varphi - 1 \wedge p.\texttt{oneLdr} \mapsto p.\texttt{status} \leftarrow \texttt{Out}; p.\langle\texttt{deliver}\rangle(p.\texttt{me})$

Complexity of $\mathcal{S}_{\mathcal{GSLE}}$. The time complexity of a snap-stabilizing algorithm is measured in terms of *delay*, *response time*, and *service time*. The delay corresponds to the maximum time, starting from any configuration, before any process p starts a service ($p.\texttt{status} \leftarrow \texttt{Wait}$). The response time is the maximum time between the start of a service ($p.\texttt{status} \leftarrow \texttt{Wait}$) and its completion ($p.\texttt{status} \leftarrow \texttt{Out}$), which also corresponds to the delivery of the result. The service time is the sum of the delay and the response time. Since $\mathcal{S}_{\mathcal{GSLE}}$ is probabilistic, we give expected values for all these measures.

Assume that an application at process p needs a service, *i.e.*, $p.\langle\texttt{need}\rangle()$ is $true$, and that the application is continuously requesting until p starts the service. If $p.\texttt{status} = \texttt{Out}$, p starts immediately. If $p.\texttt{status} = \texttt{In}$, the start is delayed until the completion of the current service (until $p.\texttt{status} \leftarrow \texttt{Out}$). If $p.\texttt{status} = \texttt{Wait}$, p should first switch $p.\texttt{status}$ from `Wait` to `In` and then from `In` to `Out` before starting. This means that the *delay*, *response time*, and *service time* are of the same order of magnitude, and depend on:

T1: *The time p spends before switching $p.\texttt{status}$ from* `Out` *to* `Wait`*, when* $p.\langle\texttt{need}\rangle() = true$: 1 round.

T2: *The time p spends before switching $p.\texttt{status}$ from* `Wait` *to* `In`: at most $\texttt{CMax} + \xi - 1 = 12B - 1$ rounds.

T3: *The expected time p spends before switching $p.\texttt{status}$ from* `In` *to* `Out`.

Time **T3** is bounded by the stabilization time ST of $\mathcal{U}$ plus the length ξ of one synchronized cycle (at the first configuration satisfying P_{SU}, a cycle may already be in progress), plus the expected number of rounds used to elect a leader once synchronization is achieved. This latter number is equal to the product of ξ and the expected number EC of synchronized cycles that should be executed to elect a leader. Overall, **T3** is less or equal to $ST + (1 + EC)\xi$ rounds with $ST \leq 6B$ and $\xi = 6B$.

The value EC depends on α_{vote}. The random choices made in each cycle are independent, so the number of synchronized cycles needed to elect a leader follows a geometric distribution with parameter PE, where PE is the probability of electing a leader in one cycle; and $EC = \frac{1}{PE}$. Now, if exactly one process randomly chooses $true$ in its variable `me` during a synchronized cycle, then a leader is elected during that cycle. So, the probability $P1$ of that event satisfies $P1 \leq PE$, and consequently, $EC \leq \frac{1}{P1}$. Furthermore, we can show that $P1 = n\alpha_{vote}(1-\alpha_{vote})^{n-1}$, and $P1$ is maximal for $\alpha_{vote} = \frac{1}{n}$. Considering the knowledge of processes (*i.e.*, $B < n \leq 2B$), the best choice for α_{vote} is in $[\frac{1}{2B}, \frac{1}{B+1}]$. In that case, we can show that $\frac{1}{P1} \leq \frac{e^2}{2}$, where e is the Euler constant. So, $EC \leq \frac{e^2}{2} \leq 3.70$ and we can conclude that the expected delay, response time, and service time are all $O(B)$ rounds, *i.e.*, $O(n)$ rounds.

Theorem 1 *$\mathcal{S}_{\mathcal{GSLE}}$ is probabilistically snap-stabilizing w.r.t. $\mathcal{S}_{gs,leader}$ in anonymous synchronous networks. If $\alpha_{vote} \in [\frac{1}{2B}, \frac{1}{B+1}]$, the expected delay, response, and service times of $\mathcal{S}_{\mathcal{GSLE}}$ are each $O(n)$ rounds.*

4.2 Asynchronous Settings

We now assume an unfair daemon: $\mathcal{S}_{\mathcal{GSLE}}$ should be adapted to take asynchrony into account, and we call the new version $\mathcal{A}_{\mathcal{GSLE}}$. As before, $\mathcal{A}_{\mathcal{GSLE}}$ is made of the same three modules, which are slightly modified. The module `Pulse` uses the same unison algorithm $\mathcal{U}$, but the parameters and the observers are adapted to handle asynchrony. The module `Election` is almost the same (in particular $\varphi = 2B$), but the reading of shared variables must be carefully managed in order to emulate the synchronous executions of the `Election` cycles. Finally, the module `Service` is left unchanged.

Asynchronous Unison. In asynchronous settings, the strict clock synchronization can no longer be assumed. The asynchronous version of the unison specification is relaxed as follows: the clocks of every two neighboring processes should not differ from more than one, and each process should increment its clock infinitely often. In [7], $\mathcal{U}$ is proven to be self-stabilizing for this specification under an unfair daemon in any anonymous network if $\kappa \geq n-2$ and $\xi \geq n+1$ (the same requirement as for the synchronous unison). We use the same value of κ as in $\mathcal{S}_{\mathcal{GSLE}}$, *i.e.*, $2B-2$. Then, note that in [5] (Theorem 61, page 104), $\mathcal{U}$ is proven to stabilize in at most $n+\kappa \leq 4B$ rounds. Moreover, its legitimate configurations are defined to be all configurations satisfying $P_{AU} \stackrel{\texttt{def}}{=} \forall p,q \in V, |p.\texttt{clock} - q.\texttt{clock}| \leq 1 \wedge p.\texttt{clock} \geq 0$.

Emulate Synchronous Cycles. Assume that the system is in a configuration satisfying P_{AU}. When a process p increments $p.\texttt{clock}$ from x to $\Phi(x)$, each neighbor q of p has either the same clock value ($q.\texttt{clock} = x$) or is one tick ahead ($q.\texttt{clock} = \Phi(x)$). In the former case, p can execute a step of `Election`, making use of the current local state of q. In the latter case, p should make use of the previous state of q (when $q.\texttt{clock}$ was equal to x). In order to do so, we modify the reading of variables of `Election` as follows. Each process p is now equipped with an additional vector variable $p.\texttt{prev}$, in which, at each clock increment, it saves its current local state *w.r.t.* Module `Election` before making any *writing* in that module. Furthermore, each direct *reading* of some process p to any `Election` variable v of one of its neighbors q is replaced by a call

to the function $p.\texttt{read}(v, q)$, which returns $q.v$ if $p.\texttt{clock} = q.\texttt{clock}$; $q.\texttt{prev}.v$, otherwise.

Local Observers. We also modify the observers in `Pulse` since (1) waiting for $\texttt{CMax} = 6B$ local steps is no longer sufficient to guarantee that the system has reached a configuration satisfying P_{AU}; and (2) the result of a full cycle can be guaranteed only if all processes have reached some synchronization barrier. We can show that this barrier is reached after any process increments its clock at least $4\mathcal{D}$ times from any legitimate configuration.

We could used the upper bound given in [12] to ensure that the system reached a configuration satisfying P_{AU}. However, this bound is in $\Theta(\mathcal{D}n^3)$ steps. So, this would drastically impact the time complexity of our algorithm.

Instead, we borrow the ideas given in [6] by using the following result: If $p.\texttt{clock}$ successively takes values $u, u+1, ..., u+(2\mathcal{D}+1)$ between configurations γ_{t_0} and $\gamma_{t_{2\mathcal{D}+1}}$ with $\forall i \in \{1, ..., 2\mathcal{D}+1\}$, $u + i > 0$, then every other process executes at least one step between configurations γ_{t_0} and $\gamma_{t_{2\mathcal{D}+1}}$.

This result provides a mechanism allowing a process to locally observe whether at least one round has elapsed. Indeed, by definition, if a process observes that all processes execute at least one step, then at least one round has elapsed. So, to decide that the configuration satisfies P_{AU}, a process p should observe that (i) all processes execute at least $n + \kappa$ rounds (the actual stabilization time of $\mathcal{U}$ for the asynchronous specification). In addition, (ii) p should increment its clock at least $4\mathcal{D}$ times after these rounds. So, it is sufficient that p counts $(2\mathcal{D}+1) \times (n+\kappa) + 4\mathcal{D} \leq 16B^2 + 12B$ *consecutive positive (local) increments* to ensure both (i) and (ii).

We have modified the local observers by first respectively setting ξ and `CMax` to $16B^2+12B+1$ and $16B^2+12B$. The function $\texttt{init}()$ and $\texttt{OK}()$ remain unchanged, but the way `cnt` is modified is slightly more complex. Indeed, when `cnt` reaches 0, `CMax` consecutive positive increments of the clock must have occurred. Now, $p.\texttt{clock}$ may become non-positive for two reasons. (1) During the stabilization phase of $\mathcal{U}$, p may set $p.\texttt{clock}$ to a negative value; or (2) p may "normally" reset $p.\texttt{clock}$ from $\xi - 1$ to 0. To handle Case (1), p also resets $p.\texttt{cnt}$ to `CMax` each time it sets $p.\texttt{clock}$ to a negative value (*i.e.*, we add $p.\texttt{cnt} \leftarrow \texttt{CMax}$ in the statement of the reset action of $\mathcal{U}$) and p does not decrement $p.\texttt{cnt}$ while $p.\texttt{clock}$ is negative. For Case (2), p starts decrementing $p.\texttt{cnt}$ only when $p.\texttt{clock} = 0$; since $\xi > \texttt{CMax}$, the case cannot occur. Hence, at each local tick, $p.\texttt{cnt}$ is decremented only if $p.\texttt{cnt} > 0 \wedge (p.\texttt{cnt} = \texttt{CMax} \Rightarrow p.\texttt{clock} = 0)$.

Complexity of $\mathcal{A}_{\mathcal{GSLE}}$. As in the synchronous version $\mathcal{S}_{\mathcal{GSLE}}$, the delay, response time, and service time of $\mathcal{A}_{\mathcal{GSLE}}$ are of the same order of magnitude, and depend on **T1**, **T2**, and **T3**, as defined in Section 4.1. Again, we assume that when at some process p, $p.\langle\texttt{need}\rangle()$ is continuously $true$ until p starts the service. So, **T1** is performed in at most one round. Then, to evaluate **T2** and **T3**, we should remark that, contrary to the synchronous case, a process does not advance its local clock at every round, even when the asynchronous unison specification is achieved. However, we can use the lemma given in [11], which claims that once P_{AU} holds, every process advances its clock at least $\mathcal{D}$ ticks during any $2\mathcal{D}$ consecutive rounds.

Consider **T2**. In the worst case, we need that $\mathcal{U}$ first stabilizes. Then, p may have to advance its local clock at most $\xi - 1$ ticks before starting to decrement $p.\texttt{cnt}$. This

latter counter reaches 0 at most CMax ticks later. Finally, once $p.\texttt{cnt} = 0$, p may have to advance its local clock up to $\xi - 1$ times before satisfying $p.\texttt{clock} = 3\varphi - 1$ and executing $p.\texttt{status} \leftarrow \texttt{Out}$ at its next tick. Hence, this complexity is bounded by $n + \kappa + \frac{2\mathcal{D}(\texttt{CMax}+2\xi-1)}{\mathcal{D}} = O(n^2)$ rounds.

Consider **T3**. As in the synchronous case, this expected time is bounded by the stabilization time of $\mathcal{U}$ plus the product between $EC + 1$ and the number of rounds to execute a synchronized cycle. Just as in the synchronous case, the best choice is to choose α_{vote} in $[\frac{1}{2B}, \frac{1}{B+1}]$. In this case, we still have $EC \leq \frac{e^2}{2} \leq 3.70$. Moreover, each synchronized cycle is executed in at most $2\xi = O(n^2)$ rounds. Hence, the time complexity of **T3** is also $O(n^2)$ rounds, and we have the following result.

Theorem 2 *With an unfair daemon, $\mathcal{A}_{\mathcal{GSLE}}$ is probabilistically snap-stabilizing w.r.t. $\mathcal{S}_{gs,leader}$ for anonymous networks. If $\alpha_{vote} \in [\frac{1}{2B}, \frac{1}{B+1}]$, its expected delay, response, and service times are each $O(n^2)$ rounds.*

If we add the assumption that processes know an upper bound D on the diameter $\mathcal{D}$ of the network, then $\mathcal{A}_{\mathcal{GSLE}}$ can be modified so that its expected time complexities are reduced to $O(D.n)$ rounds, by setting ξ and CMax to $8DB + 4B + 4D + 1$ and $8DB + 4B + 4D$, respectively.

5 Conclusion

We have introduced probabilistic snap-stabilization. Our goal is to relax (deterministic) snap-stabilization without altering its strong safety guarantees. We adopt a Las Vegas approach: after the end of faults, a probabilistic snap-stabilizing algorithm immediately satisfies its safety property; whereas its liveness property is ensured with probability 1. (It could be worth investigating if the Monte Carlo approach can be interesting.)

We implement this new concept in two algorithms which solve the guaranteed service leader election in arbitrary anonymous networks, a problem having neither self- nor snap deterministic solutions. Our first algorithm assumes a synchronous daemon, while the second works under a distributed unfair daemon, the weakest scheduling assumption. Note that these two algorithms are also self-stabilizing for the leader election problem.

These two algorithms show that probabilistic snap-stabilization is more expressive than its deterministic counterpart. Note that one can easily modify our guaranteed service leader election to obtain a guaranteed service algorithm whose result is the guarantee that the whole network has been identified. Then, using this algorithm, we can mimic the behavior of an identified network and emulate the transformer proposed in [9]. As a consequence, every (non-stabilizing) algorithm that can be made (deterministically) snap-stabilizing in an identified network by the transformer of [9] can be also automatically turned into probabilistic snap-stabilizing guaranteed service algorithm working in an anonymous network.

References

1. Alpern, B., Schneider, F.B.: Defining liveness. Inf. Process. Lett. 21(4), 181–185 (1985)

2. Angluin, D.: Local and global properties in networks of processors (extended abstract). In: 12th Annual ACM Symposium on Theory of Computing. pp. 82–93. ACM (1980)
3. Beauquier, J., Genolini, C., Kutten, S.: k-stabilization of reactive tasks. In: PODC. p. 318 (1998)
4. Beauquier, J., Gradinariu, M., Johnen, C.: Randomized self-stabilizing and space optimal leader election under arbitrary scheduler on rings. Dist. Comp. 20(1), 75–93 (2007)
5. Boulinier, C.: L'unisson. Ph.D. thesis, Université de Picardie Jules Verne (2007)
6. Boulinier, C., Levert, M., Petit, F.: Snap-stabilizing waves in anonymous networks. In: ICDCN. pp. 191–202 (2008)
7. Boulinier, C., Petit, F., Villain, V.: When graph theory helps self-stabilization. In: PODC. pp. 150–159 (2004)
8. Bui, A., Datta, A.K., Petit, F., Villain, V.: Snap-stabilization and PIF in tree networks. Dist. Comp. 20(1), 3–19 (2007)
9. Cournier, A., Datta, A.K., Petit, F., Villain, V.: Enabling snap-stabilization. In: ICDCS. pp. 12–19 (2003)
10. Cournier, A., Devismes, S., Villain, V.: Snap-stabilizing pif and useless computations. In: ICPADS. pp. 39–48 (2006)
11. Datta, A., Larmore, L., Devismes, S., Heurtefeux, K., Rivierre, Y.: Self-stabilizing small k-dominating sets. International Journal of Networking and Computing 3(1) (2013)
12. Devismes, S., Petit, F.: On efficiency of unison. In: TADDS. pp. 20–25 (2012)
13. Devismes, S., Tixeuil, S., Yamashita, M.: Weak vs. self vs. probabilistic stabilization. In: ICDCS. pp. 681–688 (2008)
14. Dijkstra, E.W.: Self-Stabilizing Systems in Spite of Distributed Control. Commun. ACM 17, 643–644 (1974)
15. Dolev, S., Israeli, A., Moran, S.: Uniform Dynamic Self-Stabilizing Leader Election. IEEE Trans. Parallel Distrib. Syst. 8, 424–440 (1997)
16. Dolev, S., Herman, T.: Superstabilizing protocols for dynamic distributed systems (abstract). In: PODC. p. 255 (1995)
17. Duflot, M., Fribourg, L., Picaronny, C.: Randomized finite-state distributed algorithms as markov chains. In: DISC. pp. 240–254 (2001)
18. Durrett, R.: Probability, theory and examples. Cambridge (2010)
19. Ghosh, S., Gupta, A., Herman, T., Pemmaraju, S.V.: Fault-containing self-stabilizing algorithms. In: PODC. pp. 45–54 (1996)
20. Gouda, M.G.: The theory of weak stabilization. In: WSS. pp. 114–123 (2001)
21. Herman, T.: Probabilistic self-stabilization. Inf. Proc. Letters 35(2), 63–67 (1990)
22. Israeli, A., Jalfon, M.: Token management schemes and random walks yield self-stabilizing mutual exclusion. In: PODC. pp. 119–131 (1990)
23. Manna, Z., Pnueli, A.: A hierarchy of temporal properties. In: PODC. pp. 377–410 (1990)
24. Matias, Y., Afek, Y.: Simple and efficient election algorithms for anonymous networks. In: WDAG. pp. 183–194 (1989)
25. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley & Sons, Inc., 1st edn. (1994)
26. Tel, G.: Introduction to Distributed Algorithms. Cambridge University Press, 2nd edn. (2001)
27. Yamashita, M., Kameda, T.: Computing on anonymous networks: Part i-characterizing the solvable cases. IEEE Trans. Parallel Distrib. Syst. 7(1), 69–89 (1996)