Automated Analysis of Impact of Scheduling on Performance of Self-Stabilizing Protocols*

Saba Aflaki[1], Borzoo Bonakdarpour[2], and Sébastien Tixeuil[3]

[1] School of Computer Science
University of Waterloo
200 University Ave West, Waterloo, ON N2L 3G1, Canada
Email: saflaki@uwaterloo.ca
[2] Department of Computing and Software
McMaster University
1280 Main Street West, Hamilton, ON L8S 4L7, Canada
Email: borzoo@mcmaster.ca
[3] Université Pierre & Marie Curie
Institut Universitaire de France
Email: sebastien.tixeuil@lip6.fr

Abstract. In a concurrent computing system, a *scheduler* determines at each time which computing task should execute next. Thus, a scheduler has tremendous impact on the performance of the tasks that it orchestrates. Analyzing the impact of scheduling in a *distributed* setting is a challenging task, as it is concerned with subtle dimensions such as geographical distance of processes and the achievable level of parallelism. In this paper, we propose an automated method based on probabilistic verification for analyzing fault recovery time in distributed *self-stabilizing* protocols. We exhibit the usefulness of our approach through a large set of experiments that demonstrate the impact of different types of scheduling policies on recovery time of different classes of stabilizing protocols, and the practical efficiency of classical self-stabilizing scheduler transformers.

1 Introduction

Self-stabilization [6] is a versatile technique for forward recovery to a good behavior when transient faults occur in a distributed system or the system is initialized arbitrarily. Moreover, once the good behavior is recovered, the system preserves this behavior in the absence of faults. In [8, 9], we demonstrated that *expected recovery time* is a more descriptive metric than the traditional asymptotic complexity measures (e.g., the big O notation for the number of recovery steps or rounds) to characterize the performance of stabilizing programs. Average recovery time can be measured by giving weights to states and transitions of a stabilizing program and computing the expected value of the number of steps

* This is an extended version of the paper appeared in the proceedings of SSS'15.

that it takes the program to reach a legitimate state. These weights can be assigned by a uniform distribution (in the simplest case), or by more sophisticated probability distributions. This technique has been shown to be effective in measuring the performance of weak-stabilizing programs, where not all computations converge [8], and cases where faults hit certain variables or locations more often, as well as in synthesizing stabilizing protocols [1].

A vital factor in designing self-stabilizing protocols is the scheduling assumptions. For instance, certain protocols are stabilizing under a (1) fair scheduler [5], (2) probabilistic scheduler [13, 17], or (3) scheduler that disallows fully parallel execution of processes [11]. In addition to the issue of correctness, different scheduling policies may have a totally different impact on the *performance* of a self-stabilizing protocol [4]. To the best of our knowledge, there is no work on rigorous analysis of how a scheduling policy alters the performance of self-stabilization.

With this motivation, in this paper, we extend our work in [8, 9] to incorporate different scheduling policies in evaluating the performance of self-stabilizing algorithms. In particular, we consider the following scheduling criteria:

- *Distribution* imposes spatial constraints on the selection of processes whose transitions will be executed.
- *Boundedness* of a scheduler ensures that a process is not scheduled for execution more than certain number of times between any two schedulings of any other process.
- *Fairness* of a scheduler guarantees that every process is given a fair share of execution.

Our contributions in this paper are the following: We

- rigorously formalize the aforementioned scheduling criteria;
- propose an automated compositional method for (1) augmenting a self-stabilizing protocol with different types of scheduling policies, and (2) evaluating the performance (*i.e.*, expected recovery time) of the augmented protocol using probabilistic verification techniques, and
- conduct a large set of experiments to demonstrate the application of our approach in analyzing the performance of self-stabilizing protocols under different scheduling policies.

  In particular, we studied the effect of distribution, boundedness and fairness of a scheduler on the possibility of convergence as well as the expected recovery time of three self-stabilizing algorithms [11], and a few variants. We show that the first algorithm needs refinement to be able to stabilize under weaker scheduling constraints. One approach is to compose the algorithm with a stabilizing local mutual exclusion algorithm. In this regard, we compose the algorithm with a snap-stabilizing dining philosophers algorithm [14] and demonstrate that ensuring safety comes at a cost of higher expected recovery time. Another approach suggested in [11,12] is to randomize the actions of processes for which we explore three different strategies for

choosing the randomization parameter (two static and one dynamic). Furthermore, we consider the solution of [11] for an identified network. This algorithm is deterministic.
We measure the expected recovery time of all these six strategies under different scheduling constraints. Our experiments show that, in general, the deterministic algorithms outperform the randomized ones. Moreover, an adaptive randomized algorithm which dynamically chooses the randomization parameter has a promising performance. It also has the benefit of no pre-tuning requirements.

Organization The rest of the paper is organized as follows. Section 2 presents our computation model for distributed programs, the concept of self-stabilization and recovery time. We present the formal semantics of different types of schedulers in Section 3. Our approach for augmenting a distributed program with a scheduling scheme is presented in Section 4. Experimental results and analysis are discussed in Section 5. Finally, we make concluding remarks and discuss future work in Section 6.

2 Preliminaries

2.1 Distributed Programs

A distributed system consists of a finite set of processes Π operating on a finite set of variables V. Each variable $v \in V$ has a finite domain D_v. A valuation of all variables determines a state of the system. We denote the value of a variable v in state s by $v(s)$. The *state space* of a distributed system is the set of all possible states spanned by V denoted by S_V.

Each process $\pi \in \Pi$ can read (respectively, write) a subset of V called its read (respectively, write)-set. We denote the read and write sets of a process by R_π and W_π respectively. In our model, processes communicate through *shared memory*, i.e. several processes can read the same variable. However, only one process can write to a variable. We say that two processes π and π' are *neighbors* if $R_\pi \cap R_{\pi'} \neq \emptyset$. Thus, the communication network of a distributed system can be modelled by a graph $G = (\Pi, E)$, where a vertex represents a process, and there is an edge between any two processes that are neighbors. We denote the shortest path between two vertices (processes) π and π' in G by $dist(\pi, \pi')$ and the diameter of the graph by $diam(G)$.

Definition 1. *A* distributed program *is a tuple* $dp = (\Pi, V, T)$*, where*

- Π *is a finite set of processes,*
- V *is a finite set of variables,*
- $T \subseteq S_V \times S_V$ *is the* transition relation. □

In order to analyze the performance of distributed programs, we view them as a *discrete-time Markov chain* (DTMC).

Definition 2. *A* DTMC *is a tuple* $M = (S, S_0, \iota_{init}, \mathbf{P_M}, L, AP)$ *where,*

- S *is a finite set of states,*
- S_0 *is the set of initial states,*
- $\iota_{init} : S \to [0, 1]$ *is the initial distribution such that* $\sum_{s \in S} \iota_{init}(s) = 1$*,*
- $\mathbf{P_M} : S \times S \to [0, 1]$ *is the transition probability matrix (TPM) such that*

$$\forall s \in S : \sum_{s' \in S} \mathbf{P_M}(s, s') = 1$$

- $L : S \to 2^{AP}$ *is the labelling function that identifies which atomic propositions from a finite set* AP *hold in each state.* □

Given Def. 1 and Def. 2, it is straightforward to model the transition system of a distributed program with a DTMC. The state space of the distributed program forms the set of states DTMC (i.e., $S = S_V$). S_0 and ι_{init} can be determined based on the program. If the distributed program is probabilistic, then the value of the elements of $\mathbf{P_M}$ are known. Otherwise, without loss of generality, we can consider uniform distribution over transitions. In that case:

$$\mathbf{P_M}(s, s') = \frac{1}{|\{(s, s'') \in T(dp)\}|}$$

L assigns atomic propositions to states which facilitates computation and verification of certain quantitative and qualitative properties. Later, in Section 2.2, we will see how a single atomic proposition ls can define an important class of distributed programs namely, self-stabilizing programs. Throughout this paper, We use dp and M interchangeably to refer to a distributed program.

Definition 3. *A* computation σ *of a distributed program* $dp = (\Pi, V, T)$ *(respectively, DTMC* $M = (S, S_0, \iota_{init}, \mathbf{P_M}, L, AP)$*), over state space* S*, is a maximal sequence of states:* $\sigma = s_0 s_1 s_2 \cdots$*, where*

- $s_0 \in S_0$*,*
- $\forall i \geq 0$*,* $(s, s') \in T(dp)$ *(respectively,* $\mathbf{P_M}(s_i, s_{i+1}) > 0$*).* □

Notation 1 σ_n *indicates a finite computation of length* n *and* σ_s*, a computation that starts in state* s*. We denote the set of all possible distributed programs by* DP*, the set of all computations of a distributed program by* $\Sigma(dp)$ *and the set of finite computations by* $\Sigma_{fin}(dp)$*.*

In the sequel, we review *cylinder sets* and *reachability probabilities* in Markov chains from [3].

Definition 4. *The* cylinder set *of a finite computation* σ_n *is the set of infinite computations which start with* σ_n*:* $Cyl(\sigma_n) = \{\sigma \mid \sigma_n \in prefix(\sigma)\}$ □

The probability of a cylinder set is given by:

$$Pr(Cyl(s_0 \cdots s_n)) = \iota_{init}(s_0) \prod_{0 \leq i < n} \mathbf{P_M}(s_i, s_{i+1}) \tag{1}$$

Reachability probability is a common quantitative property measured in Markov chains. Given a subset B of the state space S, we look for the probability of eventually reaching a state $s \in B$ (denoted by $(\Diamond)$). In other words, we are interested in computations with the initial sequence of states of the form $s_0 \cdots s_{n-1} \notin B$ and $s_n \in B$. All such computations can be indicated by the regular expression $R_B(M) = \Sigma_{fin}(M) \cap (S \backslash B)^* B$. Hence, reachability probability can be computed as follows:

$$Pr(\Diamond B) = \sum_{s_0 \cdots s_n \in \Sigma_{fin} \cap (S \backslash B)^* B} \iota_{init}(s_0) \prod_{0 \leq i < n} \mathbf{P_M}(s_i, s_{i+1}) \tag{2}$$

2.2 Self-stabilization and Convergence Time

A distributed program is *self-stabilizing* if (1) starting from an arbitrary initial state, all computations reach a state in the set of *legitimate states* (LS) in a finite number of steps without outside intervention, (2) after which remains there in the absence of faults. The first condition is known as *strong convergence* and the second as *closure*. An arbitrary state can be reached due to erroneous initialization or transient faults.

Definition 5 (self-stabilization). *A distributed program* $M = (S, S_0 = S, \mathbf{P_M}, L, \{ls\})$ *is* self-stabilizing *iff the following conditions hold:*

- *Strong convergence:* $\forall s \in S$, all *computations* σ_s *eventually reach a state in* $LS = \{s \mid ls \in L(s)\}$,
- *Closure:* $\forall s \in LS : (\mathbf{P_M}(s, s') > 0) \Rightarrow (s' \in LS)$. □

It has been shown that some problems do not have a self-stabilizing solution [12]. Thus, the strong convergence property has been relaxed in other variants of stabilization to tackle the impossibility issues and to reduce resource consumption. *Weak-stabilization* [10] ensures the possibility of convergence and *probabilistic-stabilization* [13] guarantees convergence with probability one.

Definition 6 (weak-stabilization). *A distributed program* $M = (S, S_0 = S, \mathbf{P_M}, L, \{ls\})$ *is* weak-stabilizing *iff the following conditions hold:*

- *Weak convergence: For all* $s \in S$, *there* exists *a computation* σ_s *that eventually reaches a state in* LS,
- *Closure:* $\forall s \in LS : (\mathbf{P_M}(s, s') > 0) \Rightarrow (s' \in LS)$. □

Definition 7 (probabilistic-stabilization). *A distributed program* $M = (S, S_0 = S, \mathbf{P_M}, L, \{ls\})$ *is* probabilistic-stabilizing *iff the following conditions hold:*

– *Probabilistic convergence: For all $s \in S$, a computation σ_s reaches a state in LS with probability one,*
– *Closure:* $\forall s \in LS : (\mathbf{P_M}(s, s') > 0) \Rightarrow (s' \in LS)$. □

In the sequel, we use the term *stabilizing* algorithm to refer to either of the three types of stabilization mentioned above. The expected convergence (recovery) time of a stabilizing algorithm is a key measure of its efficiency [9]. To define this metric, we use the concept of cylinder sets (Def. 4) and the reachability probability (Eq. 2) from Section 2.1.

The first time a computation of a stabilizing program reaches LS ($B = LS$) gives us the recovery time of that computation. More formally,

Definition 8. *For a stabilizing program M, the* convergence time *or* recovery time *of a computation σ with an initial fragment $s_0 s_1 \cdots s_n$ such that $s_0 s_1 \cdots s_{n-1} \notin LS$ and $s_n \in LS$ equals n.*

The expected recovery time of a stabilizing program is the expected value of the recovery time of all states in S. $\iota_{init}(s)$ indicates the probability of starting the program in state s at time 0. If this initial distribution is unknown, we can consider a uniform distribution $\iota_{init}(s) = \frac{1}{|S|}$ for all $s \in S$. Given Def. 8 and Eq. 1, the expected recovery time (ERT) of a stabilizing program M is derived by the following equation:

$$ERT(M) = \sum_{s_0 \in S_0} \sum_{\substack{\sigma_n \in R_B(M) \\ 0 \leq n < \infty}} n.\iota_{init}(s_0). \prod_{0 \leq i < n} \mathbf{P_M}(s_i, s_{i+1}) \tag{3}$$

3 Scheduler Types

Schedulers determine the degree of parallelism in a distributed program. They are specifically important in stabilizing programs as they affect both the possibility of convergence and convergence time. A detailed survey of schedulers in self-stabilization can be found in [7]. In this section, we review the four classification factors of schedulers that we consider in this paper, namely *distribution*, *fairness*, *boundedness* and *enabledness* studied in [7].

A *scheduler* d is a function that associates to each distributed program a subset of its computations: $d : DP \rightarrow \mathcal{P}(\Sigma(dp))$, where $\mathcal{P}$ denotes the powerset. Thus, $d(dp)$ denotes the computations of dp constrained by scheduler d. A transition $t = (s, s')$ *activates* a process π in state s iff it updates at least one of the variables in W_π. *Act* associates to each transition $t = (s, s')$ the set of processes that it activates:

$$Act(s, s') = \{\pi \in \Pi \mid \exists v \in W_\pi : v(s) \neq v(s')\}$$

In a distributed program dp, we say that a process π is *enabled* in state s iff there exists a transition in dp that originates in s and activates π. Hence, the set of processes enabled in state s is given by:

$$En(s, dp) = \{\pi \in \Pi \mid \pi \text{ is enabled by dp in s}\}$$

3.1 Distribution

Distribution imposes spatial constraints on the selection of processes whose transitions will be executed.

Definition 9 (k-central). *Given a distributed system $G = (\Pi, E)$, a scheduler d is* k-central *iff:*

$$\forall dp \in DP : \forall \sigma = s_0 s_1 \cdots \in d(dp) : \forall i \in \mathbb{N} : \forall \pi_1 \neq \pi_2 \in \Pi : \\ [\pi_1 \in Act(s_i, s_{i+1}) \wedge \pi_2 \in Act(s_i, s_{i+1})] \rightarrow dist(\pi_1, \pi_2) > k \qquad \square$$

In other words, a k-central scheduler allows processes in distance at least k to execute simultaneously. In particular, 0-central and $diam(G)$-central schedulers are called *distributed* and *central* respectively. In the former, any subset of the enabled processes can be scheduled at any time. In the latter, a single process can execute at a time (i.e., the interleaving semantics).

3.2 Fairness

Schedulers are classified into four categories based on fairness assumptions. A *weakly fair* scheduler ensures that a continuously enabled process is eventually scheduled.

Definition 10 (weakly fair). *Given a distributed system $G = (\Pi, E)$, a scheduler d is* weakly fair *iff*

$$\forall dp \in DP : \forall \sigma = s_0 s_1 \cdots \in \Sigma(dp) : \\ [\exists \pi \in \Pi : \exists i \geq 0 : \forall j \geq i : (\pi \in En(s_j, dp) \wedge \pi \notin Act(s_j, s_{j+1}))] \Rightarrow \sigma \notin d(dp) \quad \square$$

A *strongly fair* scheduler ensures that a process that is enabled infinitely often is eventually scheduled.

Definition 11 (strongly fair). *Given a distributed system $G = (\Pi, E)$, a scheduler d is* strongly fair *iff*

$$\forall dp \in DP : \forall \sigma = s_0 s_1 \cdots \in \Sigma(dp) : \\ [\exists \pi \in \Pi : \exists i \geq 0 : (\forall j \geq i : \exists k \geq j : \pi \in En(s_k, dp)) \wedge (\forall j \geq i : \pi \notin Act(s_j, s_{j+1}))] \Rightarrow \\ \sigma \notin d(dp) \qquad \square$$

A third type of fairness is *Gouda fairness* which is beyond the scope of this paper. Any other scheduler that does not satisfy any type of fairness constraints is considered *unfair*.

3.3 Boundedness and Enabledness

A scheduler is *k-bounded* if it does not schedule a process more than k times between any two schedulings of any other process.

Definition 12 (k-bounded). *Given a distributed system $G = (\Pi, E)$, a scheduler d is* k-bounded *iff:*

$$\begin{aligned}\forall \pi \in \Pi : \exists k > 0 : \forall dp \in DP : \forall \sigma = s_0 s_1 \cdots \in d(dp) : \forall (i,j) \in \mathbb{N}^2 : \\ & [[\pi \in Act(s_i, s_{i+1}) \wedge (\forall l < i : \pi \notin Act(s_l, s_{l+1}))] \Rightarrow \\ & \forall \pi' \in \Pi \backslash \{\pi\} : |\{l \in \mathbb{N} \mid l < i \wedge \pi' \in Act(s_l, s_{l+1})\}| \leq k] \wedge \\ & [[i < j \wedge \pi \in Act(s_i, s_{i+1}) \wedge \pi \in Act(s_j, s_{j+1}) \wedge \\ & (\forall l \in \mathbb{N} : i < l < j \Rightarrow \pi \notin Act(s_l, s_{l+1}))] \Rightarrow \\ & \forall \pi' \in \Pi \backslash \{\pi\} : |\{l \in \mathbb{N} \mid i \leq l < j \wedge \pi' \in Act(s_l, s_{l+1})\}| \leq k] \end{aligned}$$ □

A scheduler is *k-enabled* if a process cannot be enabled more than k times before being scheduled.

Definition 13. (k-enabled) *Given a distributed system G, a scheduler d is* k-enabled *iff:*

$$\begin{aligned}\forall dp \in DP : \exists k > 0 : \forall \sigma = s_0 s_1 \cdots \in d(dp) : \forall (i,j) \in \mathbb{N}^2 : \forall \pi \in \Pi : \\ & [[\pi \in Act(s_i, s_{i+1}) \wedge (\forall l \in \mathbb{N}, l < i \Rightarrow \pi \notin Act(s_l, s_{l+1}))] \Rightarrow \\ & |\{l \in \mathbb{N} \mid l < i \wedge \pi \in En(s_l, \sigma)\}| \leq k] \wedge \\ & [[i < j \wedge \pi \in Act(s_i, s_{i+1}) \wedge \pi \in Act(s_j, s_{j+1}) \wedge \\ & (\forall l \in \mathbb{N}, i < l < j \Rightarrow \pi \notin Act(s_l, s_{l+1}))] \Rightarrow \\ & |\{l \in \mathbb{N} \mid i < l < j \wedge \pi \in En(s_l, \sigma)\}| \leq k] \wedge \\ & [[pi \in Act(s_i, s_{i+1}) \wedge (\forall l \in \mathbb{N}, l > i \Rightarrow \pi \notin Act(s_l, s_{l+1}))] \Rightarrow \\ & |\{l \in \mathbb{N} \mid l > i \wedge \pi \in En(s_l, \sigma)\}| \leq k] \end{aligned}$$ □

4 Augmenting a Distributed Program with a Scheduler

To concisely specify the behavior of a process π, we utilize a finite set of *guarded commands* ($\mathcal{G}_\pi$). A guarded command has the following syntax:

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

The *guard* is a Boolean expression over the read-set of the process. The statement is executed whenever the guard is satisfied. Execution of guarded commands updates variables and causes transitioning from one state to another. In *probabilistic programs* commands are executed with a probability. Hence, transitions among states in the system are executed according to a probability distribution.

$$\langle label \rangle \ : \ \langle guard \rangle \ \rightarrow \ p_1 : \langle statement_1 \rangle \ + \ \cdots \ + \ p_n : \langle statement_n \rangle;$$

where

$$\sum_{i=1}^{n} p_i = 1$$

A guarded command is *enabled* if its guard evaluates to true. A process is enabled if at least one of its guarded commands is enabled. The set of guarded commands of a distributed program is formed by the union of the guarded commands of its constituent processes. In a parallel (i.e., simultaneous) execution, all enabled processes execute their enabled commands. In contrast, in a serial (i.e., interleaving) execution, only one enabled process runs its enabled commands. We use labels to synchronize (parallelized) guarded commands of different processes. More specifically, if all guarded commands (possibly belonging to different processes) that have identical labels are enabled, they will all be executed. If at least one of them is not enabled, none of them will be executed. The synchronization of guarded commands with guards $g_1, \cdots, g_n$ is equivalent to having one guarded command with guard $g_1 \wedge \cdots \wedge g_n$ and the union of all statements. We omit the label from a guarded command whenever it is not used.

4.1 Encoding Schedulers in a Distributed Program

In this section, we describe how we modify a distributed program dp to obtain a program that behaves as if dp was executed under a certain type of scheduler, when only serial executions are available.

***k*-Central Scheduler** Given a distributed system composed of processes Π. Let each process π in Π consist of a set of guarded commands $\mathcal{G}_\pi$. We augment Π with a k-central scheduler as follows. For every process $\pi \in \Pi$, let

$$KValid_\pi = \{\pi' \mid dist(\pi, \pi') > k\}$$

be the set of processes that are at least $k+1$ hops away from π. To encode a k-central scheduler, we synchronize every guarded command of a process π with the guarded commands of every subset of $KValid_\pi$. Thus, each process of the new program consists of the following guarded commands:

$for\ all\ \langle g_{\pi,i}\rangle \rightarrow \langle s_{\pi,i}\rangle \in \mathcal{G}_\pi : for\ all\ kval_\pi \subseteq KValid_\pi : for\ all\ \pi' \in kval :$
$for\ all\ \langle g_{\pi',j}\rangle \rightarrow \langle s_{\pi',j}\rangle \in \mathcal{G}_{\pi'} :$
$\langle g_{\pi,i} \wedge (\bigwedge g_{\pi',j})\rangle \rightarrow \langle s_{\pi,i}\rangle;$

Note that $kval_\pi = \emptyset$ yields the original guarded command of the process. It is necessary to include this case to model a central scheduler.

***k*-Bounded and *k*-Enabled Schedulers** To simulate the behavior of a k-bounded (similarly, k-enabled) scheduler, we add a counter (variable) per every ordered pair of processes in the system. A command of a process is allowed to

execute only if it has been executed less than k times between any two executions of every other process. Once a process π executes a command, the variables which count the number of executions of other processes between any two executions of π are reset to zero. In a distributed system with n processes $\Pi = \{\pi_1, \cdots, \pi_n\}$, we add variables $\{count_{\pi_i,\pi_j} \mid 1 \leq i, j \leq n\}$. Thus, we replace each guarded command $\langle g_{\pi_i} \rangle \rightarrow \langle s_{\pi_i} \rangle$ in $\mathcal{G}_{\pi_i}$ with the following:

$$\langle g_{\pi_i} \wedge (\bigwedge_{\substack{1 \leq j \leq n \\ i \neq j}} count_{\pi_i,\pi_j} < k) \rangle \rightarrow \langle s_{\pi_i} \rangle; \{\langle count_{\pi_j,\pi_i} := 0 \rangle;\}_{\substack{1 \leq j \leq n \\ i \neq j}}$$

Fairness Schedulers that generate the worst and best cases are unfair. They can be achieved by modelling the program with a Markov decision process (MDP) instead of a DTMC (for more information see [16]). A probabilistic scheduler which uniformly chooses transitions produces average case expected recovery time, which is both fair and unfair.

5 Experiments and Analysis

We use probabilistic model-checking (in particular, the tool PRISM [15]) to investigate the significance of the choice of a scheduler on the expected recovery time of a stabilizing distributed algorithm. As discussed in Section 3, there are several factors to take into account, such as the amount of parallelism (centrality), boundedness, enabledness, fairness, and probabilistic behavior. We chose the *vertex coloring in arbitrary graphs* problem as our case study. It is a classic problem in graph theory that has many applications in scheduling, pattern matching, etc. Furthermore, we study several stabilizing programs that solve this problem. One non-probabilistic algorithm that requires a network, where each process must have a unique id. A probabilistic algorithm where a static probability is assigned to each process, and one with adaptive probability. We compare the expected recovery time of these strategies under all types of schedulers. In our experiments, the choice of graph structure/size and some other parameters was influenced and limited by the computational power of the machine used to do the experiments.

5.1 Self-stabilizing Vertex Coloring in Arbitrary Graphs

In this section, we define the vertex coloring problem, review the first self-stabilizing program from [11] and study how different schedulers affect its expected recovery time.

Definition 14 (Vertex Coloring). *In a graph $G = (V, E)$, the* vertex coloring *problem asks for a mapping from V to a set C of colors, such that no two adjacent vertices (connected directly by an edge) share the same color.* □

Algorithm 1 Deterministic Self-stabilizing Vertex Coloring Program

1: **Shared Variable:** $c_\pi : int \in [0, B]$
2: **Guarded Command:** $action_\pi : \neg(c_\pi = max(\{0, \cdots, B\} \backslash \bigcup_{\pi' \in N(\pi)} c_{\pi'})) \rightarrow c_\pi := max(\{0, \cdots, B\} \backslash \bigcup_{\pi' \in N(\pi)} c_{\pi'})$

Definition 15 (Vertex Conflict). *Two vertices are in* conflict *iff they are neighbors and they have the same color. Thus, the number of conflicts for a vertex is the number of its neighbors that have the same color as the vertex.* □

The first deterministic self-stabilizing vertex coloring program of [11] is designed for an anonymous network with an arbitrary underlying communication graph structure $G = (\Pi, E)$ and a non-distributed scheduler. We call this program *deterministic*. Each process has a variable c_π representing its color with domain $c_\pi \in [0, B]$, where B is the maximum degree (number of neighbors) of a vertex (process) in G. In every state, if a process's color is not equal to the maximum available color (the maximum number not taken by any of its neighbors) max_π, it changes its color to max_π. Otherwise, it does not do anything. In this algorithm, a legitimate state is one that the color of each process is equal to max_π. We denote the neighbors of a process by $N(\pi)$ (see Algorithm 1).

The Effect of Schedulers on Expected Recovery Time We investigate the effect of four attributes of schedulers: centrality, boundedness, enabledness, and fairness, on the expected recovery time of Algorithm 1.

***k*-Centrality:** We calculate the average case expected recovery time for a linear graph, where the size varies from $5 - 7$ and k varies from $0 - Diam(G)$. In a linear graph, $Diam(G) = size - 1$. As expected, Table 1(a) validates that, in average, parallelism helps improve the recovery time. However, there can be cases in which it shows detrimental effect. The impact of centrality also depends on the fairness of the scheduler. In the worst case this program does not stabilize under a distributed $(0 - central)$ scheduler.

Boundedness/Enabledness: We study the effect of boundedness/enabledness on graphs of size 4 with complete, star, and linear structures for $k = 1, 2, 3$. For each graph structure and each value of k, Table 1(b) contains three numbers: R_{min} (best case expected recovery time), R_{avg} (average case expected recovery time), and R_{max} (worst case expected recovery time). Table 2(a) demonstrates that as k increases so does the gap between the best case and the worst case. This is the result of allowing more computations as we increase k. That is, all executions corresponding to a k-bounded (respectively, k-enabled) scheduler are also included in the executions of a $(k-1)$-bounded (respectively, $(k-1)$-enabled) scheduler.

(a) Effect of centrality on expected recovery time (average case)

Size\k	0	1	2	3	4	5	6
5	5.6	9.1	13.1	14.4	15.7	-	-
6	7.1	10.7	14.6	18.6	19.6	20.9	-
7	7.6	11.6	16.1	21.2	24.5	24.6	25.8

(b) Effect of boundedness/enabledness on expected recovery time

	Complete			Star			Linear		
	R_{min}	R_{max}	R_{exp}	R_{min}	R_{max}	R_{exp}	R_{min}	R_{max}	R_{exp}
1	3.24	4.64	3.88	10.25	13.33	11.78	6.84	10.30	8.48
2	2.68	7.14	4.10	6.74	24.20	12.52	4.48	19.40	9.03
3	2.57	9.63	4.28	5.74	35.04	13.39	3.87	28.61	9.60

Table 1: Effect of centrality, boundedness and enabledness.

(a) $R_{max} - R_{min}$

	Complete	Star	Linear
1	1.40	3.08	3.46
2	4.46	17.46	14.92
3	7.06	29.30	24.74

(b) Cost of ensuring safety in executions

	Fair		Unfair	
Deg	deterministic 1-central	composed distributed	deterministic 1-central	composed distributed
2	1.72	2.54	2.44	2.91
3	2.29	3.73	3.52	4.75
4	2.72	4.69	4.61	6.56
5	3.05	5.49	5.69	8.34
6	3.33	6.16	6.77	10.08
7	3.56	6.72	7.82	11.77
8	3.76	7.21	8.87	13.43

Table 2

Fairness: Fairness alongside centrality can determine possibility of convergence. An unfair distributed scheduler can prevent Algorithm 1 from converging. Consider, for example, a state in which two neighbors have identical colors ($\langle 1, 1 \rangle$) and the same maximum available color (2). A computation that infinitely alternates between states $\langle 1, 1 \rangle$ and $\langle 2, 2 \rangle$ never converges to a correct state. Such a computation can be produced by a distributed unfair scheduler. In the rest of our experiments, by unfair scheduler we mean a scheduler that results in worst case expected recovery time, unless otherwise specified.

5.2 Composition with Dining Philosophers & the Cost of Ensuring Safety

Recall that Algorithm 1 needs to be refined to work under distributed unfair schedulers. We compose Algorithm 1 with an optimal *snap-stabilizing* (i.e., zero recovery time) dining philosophers distributed program for trees of [14] and refer to it as the *composed strategy*. The solution to the dining philosophers problem provides local mutual exclusion. Since this algorithm is designed specifically

for tree structures, in the rest of this section, we use balanced trees in our experiments to ensure fair comparison. Figs. 1 and 2 depict the expected recovery time of the composed algorithm under fair (central, 1-central, distributed) and unfair (central, 1-central, distributed) schedulers, respectively.

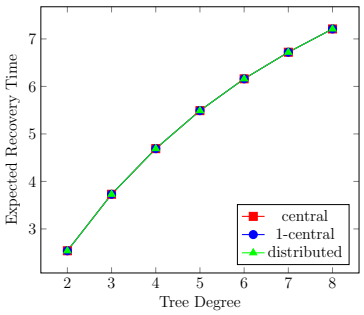

Fig. 1: Composed program with a fair scheduler

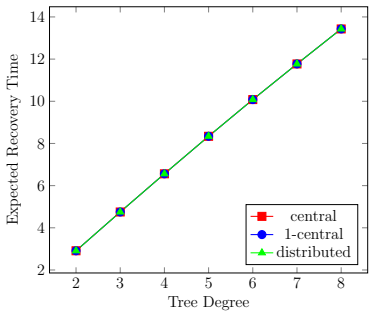

Fig. 2: Composed program with an unfair scheduler

Observe that composing a distributed program with dining philosophers and running the composition under a distributed scheduler is in principle equivalent to running the original distributed program under a 1-central scheduler. However, the 1-central scheduler that is produced by the dining philosopher algorithm may only be able to produce a subset of the possible schedules. Table 2(b) shows the expected recovery time of the deterministic algorithm under 1-central scheduler and the composed algorithm under distributed scheduler (both fair and unfair) for trees with height one and degrees $2-8$. The difference is explained by the fact that the dining philosophers layer itself forces processes that are normally not activatable (that is, they already have a non-conflicting color) to act; that is, the enforcement of fairness between nodes induces unnecessary computation steps.

5.3 ID-Based Prioritization

This strategy corresponds to the second deterministic self-stabilizing algorithm of [11], and requires an identified network where each process has a unique id. When several processes are in conflict with the same color, only the process with the highest id will execute its command. As a result, no two similarly colored enabled neighbors will ever execute their commands simultaneously. In some rare cases, this algorithm may not produce 1-central schedules: consider a line of 4 processes c, a, b, d (where identifiers are ordered alphabetically), such that c and a have the same color α, and b and d have the same color β (with $\alpha \neq \beta$). Then, a distributed scheduler may schedule both a and b in a particular step from this situation, resulting in neighboring nodes executing their actions simultaneously.

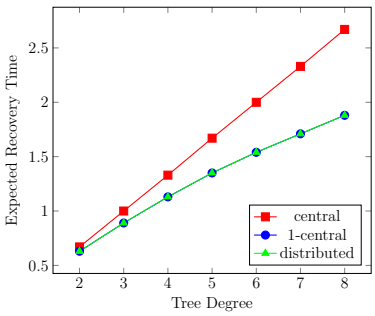

Fig. 3: ID-based deterministic program with a fair scheduler

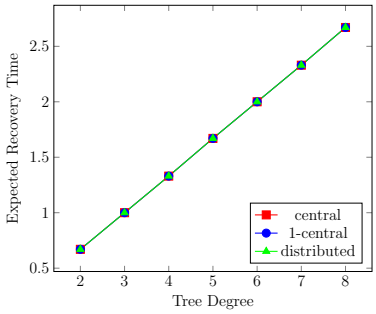

Fig. 4: ID-based deterministic program with an unfair scheduler

In trees of height 1, this situation cannot occur, and all produced schedules are 1-central. This explains in our results (see Figs. 3 and 4) why running this program under 1-central and distributed schedulers produces the same expected recovery time.

5.4 Probabilisitic-Stabilizing Vertex Coloring Programs

The random conflict manager [12] is a lightweight composition scheme for self-stabilizing programs that amounts to executing the original algorithm with some probability p (rather than always executing it). The probabilistic conflict manager does *not* ensure that two neighboring nodes are *never* scheduled simultaneously, but anytime the (possibly unfair) scheduler activates two neighboring nodes u and v, there is a $1-p^2$ probability that u and v do *not* execute simultaneously. Composing the random conflict manager with the deterministic coloring protocol yields a probabilistic coloring algorithm. Fine tuning the parameter p is challenging: a higher p reduces the possibility that a conflict persists when two neighboring conflicting nodes are activated simultaneously (reducing the stabilization time), but also reduces the possibility to make progress by executing the algorithm (increasing the stabilization time). Thus, we consider three strategies for choosing p:

1. p is a constant, for all nodes, throughout the entire execution;
2. p depends on local topology (*i.e.* the current node degree);
3. p is dynamically computed (*i.e.* depending on the current number of conflicts at the current node).

Constant Randomization Parameter In this strategy, p is a fixed constant for all processes during the program execution. In the original third probabilistic algorithm [11], this probability is equal to 0.5. Figs. 5 and 6 show that with fixed

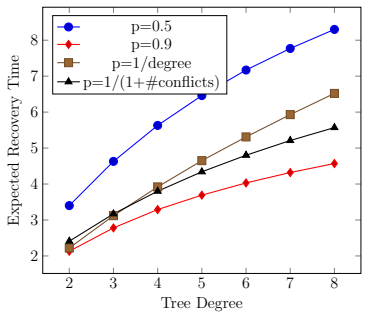

Fig. 5: Probabilistic programs with a fair distributed scheduler

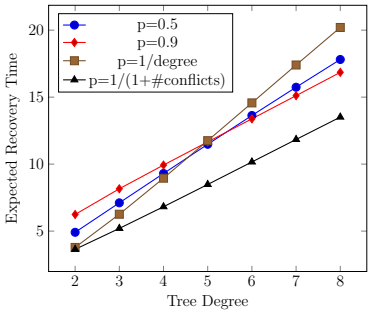

Fig. 6: Probabilistic programs with an unfair distributed scheduler

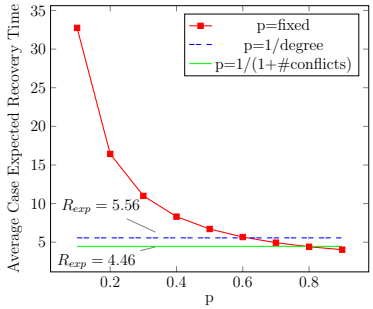

Fig. 7: Evaluating the effect of the randomization parameter on expected recovery time (on a tree of height=2, degree=2) with a fair distributed scheduler

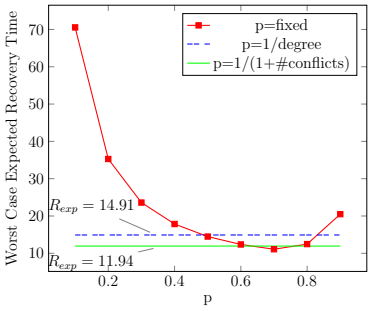

Fig. 8: Evaluating the effect of the randomization parameter on expected recovery time (on a tree of height=2, degree=2) with an unfair distributed scheduler

probability of execution, the stabilization time increases as the number of potential initial conflicts rises. Figs 7 and 8 demonstrate that for a fixed topology, fine tuning the probability used can result in significantly lower stabilization time. We observe that the stabilization time is not necessarily monotonous with respect to the probability used, as the unfair case demonstrates that increasing the probability of execution too much may have detrimental effects (more conflicts can be preserved in the worst case).

Vertex Degree This strategy depends on the local structure of the network to let a process execute its commands. It is based on the intuitive reasoning that nodes with fewer neighbors have a lower chance of being in conflict with one of them. The protocol gives higher priority to processes with less number of neighbors. Although processes can have distinct values of p, their values are

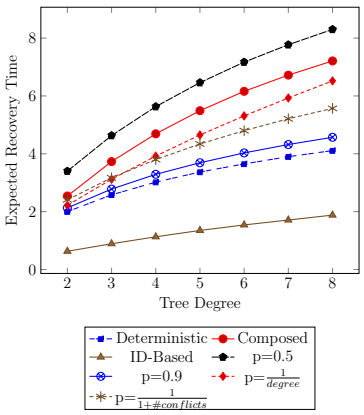

Fig. 9: Fair distributed scheduler

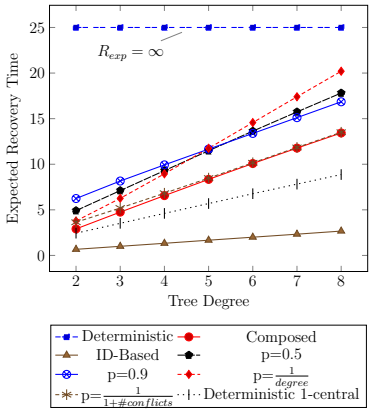

Fig. 10: Unfair distributed scheduler

statically chosen and fixed during the execution. Fig. 5 shows that this strategy works remarkably better than a fair coin under a fair scheduler. However, it gradually falls behind a fair coin in the worst case under an unfair scheduler. This is explained by the existence of a central node with an increasing number of neighbors. If executed, this central node can resolve many conflicts at the same time (expediting stabilization) in the initial case where it has many conflicts. However, the vertex degree approach pushes towards that these many conflicts are resolved by satellite nodes with a higher probability, causing stabilization to require additional steps, in the worst case.

Number of Conflicts This strategy refines the vertex degree approach to dynamically take into account the number of potential conflicts. It prioritizes processes with more conflicts over processes with fewer conflicts. Figures 5- 8 indicate that except for a few biased coins, this adaptive method defeats the other two strategies. It also has the clear advantage of no pre-tuning of the system.

5.5 Comparing Strategies & Schedulers

This section is devoted to analyzing the results of our technique to select a protocol variant for a particular environment (topology and scheduler). Figure 9 presents a comparison of protocol variants (deterministic, composed, id-based, and the three probabilistic ones) when the scheduler is fair, varying the number of nodes in the network. One interesting lesson learned is that the original protocol (deterministic), which is not self-stabilizing for the distributed scheduler (only weakly stabilizing) performs in practise better than actually self-stabilizing protocols (composed, and the three probabilistic variants), so there is a price to

pay to ensure (actual or probabilistic) self-stabilization. Overall, the id-based deterministic protocol performs the best (but requires the additional assumption that nodes are endowed with unique identifiers). We also observe that smarter probabilistic variants outperform the composed deterministic protocol, so probabilistic stabilization can come cheaper than a deterministic one.

Figure 10 describes the performance of the same protocols in the worst case (unfair scheduler). As deterministic is only weak-stabilizing, its stabilization time with unfair scheduler is infinite. In that case, all probabilistic protocols perform worse than composed, as there exists computations with longer incorrect paths of execution. We also represent the performance of deterministic under the 1-central scheduler as a reference (all other protocols are presented for the distributed scheduler) for best case situation where only 1-central execution are present. It turns out that both probabilistic variants and composed introduce overhead. The overhead of composed has been discussed in Section 5.2, while the overhead of probabilistic variants is that more executions (including executions that are not 1-central) remain possible (with respect to 1-central ones). Again, id-based outperforms all others, including those of deterministic under 1-central scheduler.

The most complex topology is presented in Fig. 11, and the relative order of strategies is preserved also for this setting. If the scheduling is fair and identifiers are not available, leaving the algorithm unchanged is the best option. Otherwise, the choice can be to use the refined probabilistic option (that is depending on the current number of conflicts) when there are no identifiers, and the id-based deterministic protocol whenever they are available.

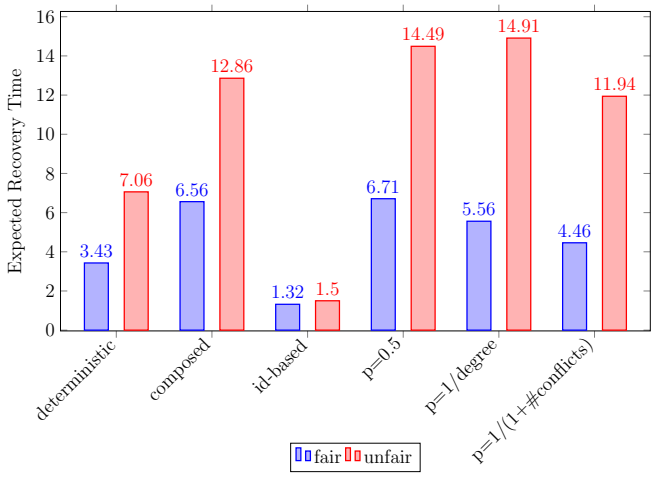

Fig. 11: Expected recovery time of the six algorithms under fair and unfair schedulers for a tree of height 2 and degree 2. Deterministic is presented for the 1-central scheduler. All others are presented for the distributed scheduler.

6 Conclusion

In this paper, we studied the role of schedulers in the correctness and performance of stabilizing programs. We adopted a rigorous method based on probabilistic model checking proven to be more descriptive by previous work [8, 9]. We investi-

gated the impact of different
scheduling criteria, namely distribution, boundedness, and fairness on the performance of the self-stabilizing vertex coloring protocols of arbitrary graphs algorithm of [11]. We explored two methods to transform the first deterministic algorithm to a scheduler-oblivious self-stabilizing program that works under distributed unfair schedulers as well. First, we composed the algorithm with a stabilizing dining philosophers algorithm at the cost of slower recovery due to the overhead induced by the additional layer (even though this layer has zero stabilization time). Second, we used a probabilistic conflict manager to ensure convergence with probability one. We studied three strategies for picking the randomization parameter p: *(i)* a constant value, *(ii)* a static value inversely proportional to the degree of the vertex (process), and *(iii)* a dynamic value inversely proportional to the number of conflicts. Our experiments establish the superiority of the final strategy, especially in the light of no tuning requirements. We also evaluated the id-based deterministic algorithm of [11] for a non-anonymous network.

In general, our results demonstrate that the deterministic algorithms outperform the probabilistic ones. The id-based algorithm is the best among all deterministic ones as well as defeating all probabilistic algorithms. The price to pay is that unique identifiers must preexist in the network. To run an id-based self-stabilizing algorithm on an anonymous network, the algorithm should be composed with a self-stabilizing unique naming algorithm. This approach, however, is likely to downgrade the performance significantly. Furthermore, precisely evaluating this performance hit requires to formally include more advanced composition techniques [2] in our framework, an interesting open challenge.

For future work, we are planning to study the impact of scheduling policy along with other factors that can affect the performance of a self-stabilizing protocol, such as the likelihood and locality of occurrence of faults.

7 Acknowledgment

This work was partially sponsored by Canada NSERC Discovery Grant 418396-2012 and NSERC Strategic Grant 430575-2012.

References

1. S. Aflaki, F. Faghih, and B. Bonakdarpour. Synthesizing self-stabilizing protocols under average recovery time constraints. In *Proceedings of the 35th International Conference on Distributed Computing Systems (ICDCS)*, 2015. To appear.
2. A. Arora, M. G. Gouda, and T. Herman. Composite routing protocols. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 70–78, 1990.
3. C. Baier and J-.P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.
4. J. Beauquier and C. Johnen. Analyze of probabilistic algorithms under indeterministic scheduler. In *IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 553–558, 2008.

5. S. Devismes, S. Tixeuil, and M. Yamashita. Weak vs. self vs. probabilistic stabilization. In *Proceedings of the 28th International Conference on Distributed Computing Systems (ICDCS)*, pages 681–688, 2008.
6. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.
7. S. Dubois and S. Tixeuil. A taxonomy of daemons in self-stabilization. *CoRR*, abs/1110.0334, 2011.
8. N. Fallahi and B. Bonakdarpour. How good is weak-stabilization? In *Proceedings of the 15th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 148–162, 2013.
9. N. Fallahi, B. Bonakdarpour, and S. Tixeuil. Rigorous performance evaluation of self-stabilization using probabilistic model checking. In *Proceedings of the 32nd IEEE International Conference on Reliable Distributed Systems (SRDS)*, pages 153 – 162, 2013.
10. M. G. Gouda. The theory of weak stabilization. In *International Workshop on Self-Stabilizing Systems*, pages 114–123. Springer, 2001.
11. M. Gradinariu and S. Tixeuil. Self-stabilizing vertex coloring of arbitrary graphs. In *Proceedings of 4th International Conference on Principles of Distributed Systems (OPODIS)*, pages 55–70, 2000.
12. M. Gradinariu and S. Tixeuil. Conflict managers for self-stabilization without fairness assumption. In *Proceedings of the 27th International Conference on Distributed Computing Systems (ICDCS)*, pages 46–46, 2007.
13. T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, 1990.
14. C. Johnen, L. O. Alima, A. K. Datta, and S. Tixeuil. Optimal snap-stabilizing neighborhood synchronizer in tree networks. *Parallel Processing Letters*, 12(3–4):327–340, 2002.
15. M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, pages 585–591, 2011.
16. A. Simaitis. *Automatic Verification of Competitive Stochastic Systems*. PhD thesis, Department of Computer Science, University of Oxford, 2014.
17. Y. Yamauchi, S. Tixeuil, S. Kijima, and M. Yamashita. Brief announcement: Probabilistic stabilization under probabilistic schedulers. In *Proceedings of the 26th International Symposium on Distributed Computing (DISC)*, pages 413–414.