Rigorous Performance Evaluation of Self-stabilization using Probabilistic Model Checking

Narges Fallahi
School of Computer Science
University of Waterloo
200 University Avenue West
Waterloo N2L 3G1, Canada
Email: nfallahi@uwaterloo.ca

Borzoo Bonakdarpour
School of Computer Science
University of Waterloo
200 University Avenue West
Waterloo N2L 3G1, Canada
Email: borzoo@cs.uwaterloo.ca

Sébastien Tixeuil
UPMC Sorbonne Universités
Institut Universitaire de France
Email: Sebastien.Tixeuil@lip6.fr

***Abstract*—We propose a new metric for effectively and accurately evaluating the performance of self-stabilizing algorithms. Self-stabilization is a versatile category of fault-tolerance that guarantees system recovery to normal behavior within a finite number of steps, when the state of the system is perturbed by transient faults (or equally, the initial state of the system can be some arbitrary state). The performance of self-stabilizing algorithms is conventionally characterized in the literature by asymptotic computation complexity. We argue that such characterization of performance is too abstract and does not reflect accurately the realities of deploying a distributed algorithm in practice. Our new metric for characterizing the performance of self-stabilizing algorithms is the expected mean value of recovery time. Our metric has several crucial features. Firstly, it encodes accurate average case speed of recovery. Secondly, we show that our evaluation method can effectively incorporate several other parameters that are of importance in practice and have no place in asymptotic computation complexity. Examples include the type of distributed scheduler, likelihood of occurrence of faults, the impact of faults on speed of recovery, and network topology. We utilize a deep analysis technique, namely, probabilistic model checking to rigorously compute our proposed metric. All our claims are backed by detailed case studies and experiments.**

***Keywords*-Self-stabilization; Performance evaluation; formal methods**

I. Introduction

Self-stabilization [8], [9], [22] is a versatile technique for forward fault recovery. When the system is hit by faults and driven to some arbitrary state [24], it is guaranteed to recover proper behavior within a finite number of execution steps. Once the system reaches such good behavior, typically specified by a set of *legitimate states*, it remains in this set thereafter in the absence of new faults. The past decades saw the initial concept maturing and spanning various problems [7], [6], [3], [1], and networks [10], [5], [18], [21].

The performance of a self-stabilizing algorithm is usually characterized by how fast it recovers good behavior. The conventional metric to evaluate the performance is asymptotic computational complexity in terms of the number of rounds [11] (*i.e.*, the shortest computation in which each process executes at least one step), or waiting time (*i.e.,* the number of global actions that has to be executed). Thus, one can express the performance of a self-stabilizing algorithm as the worst case number of, *e.g.,* rounds in terms of the big $\mathcal{O}$ notation of a parameter of the distributed system. For instance, if the topology of a distributed system is a tree, then the performance could be a polynomial of h rounds, where h is the height of the tree.

We argue that such characterization of performance is too abstract and does not reflect accurately the realities of deploying a distributed self-stabilizing algorithm in practice. More specifically, the asymptotic computational complexity abstracts away factors that can be potentially crucial in practice (constants and smaller polynomials may matter, but most importantly the worst case complexity may be irrelevant in practice). We believe that these abstractions make asymptotic computational complexity a less attractive metric for analyzing the actual performance of self-stabilizing algorithms.

On the other hand, practical performance evaluation of self-stabilizing algorithms typically uses simulation [13], [19], [18], [1], but most approaches consider running the algorithm from an initial random state, which is known to introduce significant bias [1] and also raises the question of case coverage since only a very small portion of possible states are encountered in a typical simulation run.

With this motivation, in this paper, we propose a new metric that characterizes the average recovery speed of a self-stabilizing algorithm independently of the underlying network topology and communication technology (*e.g.,*, shared memory *vs.* message passing). The new metric is technically the statistical expected mean value of the number of recovery execution steps. This expected value is computed based on the sum of probabilities for n-step reachability of legitimate states for all possible values of n; *i.e.*, the number of execution steps to reach a legitimate state from each arbitrary state. Our technique to compute the probability of n-step reachability and subsequently the expected mean value of recovery speed is based upon automated push-button probabilistic model checking [23]. A probabilistic model checker takes the model of a (deterministic or non-

deterministic) system and a set of probabilistic temporal properties as input and automatically proves whether the model satisfies the properties. In the context of our problem, a probabilistic temporal property is of the form 'whether the probability of reachability of legitimate states from an arbitrary state within n execution steps is greater than a certain value'. Equally, one can compute the probability of n-reachability of legitimate states from an arbitrary state, which modern probabilistic model checkers can do. Obviously, computing this probability implies the computation of the expected mean value of recovery speed. The average recovery speed computed using this technique represents the overall performance of an algorithm and can be used as a basis for comparing the performance of different algorithms proposed to solve the same problem in a more realistic fashion. We note that our method is scalable proportional to probabilistic model checking.

Another advantage of our technique is that it can elegantly take into account other parameters vital to evaluation of the performance of a self-stabilizing algorithm by simply incorporating them when building the model of an algorithm. Examples of these parameters include the impact of underlying scheduler and the likelihood of occurrence of faults. While the former can significantly change the way an algorithm behaves during recovery, the latter determines the number of states and the probability of reaching states from where recovery is fast or slow. This number and probability can significantly impact the expected mean value of the speed of recovery. To the best of our knowledge, such analysis has not been addressed in the literature of self-stabilization.

In order to back our claims, we conducted thorough experiments using the PRISM model checker [16] and three well-known self-stabilizing algorithms for solving distributed Propagation of Information with Feedback (PIF) in rooted trees. Our experiments clearly show that an algorithm that has the best asymptotic performance does not perform the best under all possible scenarios. Thus, in order to implement and deploy a self-stabilizing algorithm in practice, one has to asses the environment and likelihood of its faults to make the best choice of algorithm. In other words, our results show that the big $\mathcal{O}$ notation by itself is not a good metric for deployment of self-stabilizing algorithms in real world.

We believe that our new metric and automated technique provide valuable insights on behavior and performance of self-stabilizing algorithms crucial for practical system deployment. To demonstrate the application of such insights, we also show that the performance metric output can be of significant help in implementation, as we derive a general technique (called *state encoding*) that automatically improves (sometimes in considerable proportions) the performance of the implementation without changing the behavior of the protocol.

Organization: The rest of the paper is organized as follows. In Section II, we recap the preliminary concepts such as the formal definition of distributed systems, probabilistic model checking, and self-stabilization. Our performance evaluation method is described in Section III, while Section IV elaborates on its benefits. We analyze our experimental results in Section V. State encoding is discussed in Section VI. Finally, we make concluding remarks in Section VII.

II. Preliminaries

In this section, we describe the preliminary concepts on distributed systems, probabilistic model checking, and self-stabilization.

A. Distributed Systems

We model a *distributed system* as a simple self-loopless *static* undirected graph $\mathcal{G} = (V, E)$, where V is a finite set of vertices representing *processes* and E is a finite set of *edges* representing bidirectional communication, such that for all $(p, q) \in E$, we have $p, q \in V$. In this case, p and q are called *neighbors*. The set of all neighbors of a process p is denoted by $N(p)$.

The communication between processes are carried out using *locally shared variables*. Each process owns a finite set of locally shared variables, henceforth, referred to as *variables*. Each variable ranges over a fixed domain and the process can read and write them. Moreover, a process can also read variables of its neighbors in one atomic step. The *state* of a process is defined by a mapping from each variable to a value in its finite domain[1]. A process can change its state by executing its *local algorithm*. The local algorithm of a process is described using a finite set of Dijkstra's *guarded commands* (also called *actions*) of the form:

$$\langle label \rangle \ :: \langle guard \rangle \ \longrightarrow \ \langle statement \rangle$$

The *guard* of an action at process p is a Boolean expression involving a subset of variables of p and its neighbors. The *statement* of an action of p updates a subset of variables of p. That is, in addition to reading and writing its own variables, a process can as well read its neighbors' variables.

Example: We utilize the following as a simple running example for the sake of illustration and describing the concepts. Consider a system that consists of only one process. This process has a variable x that ranges over domain $\{0, 1, 2, 3\}$. The process actions are the following:

$$\begin{array}{lcl} x = 0 & \longrightarrow & \textbf{print}(\texttt{``safe''}) \\ x = 1 & \longrightarrow & x := 0 \\ x = 2 & \longrightarrow & x := 1 \\ x = 3 & \longrightarrow & x := 2 \\ x = 3 & \longrightarrow & x := 1 \\ x = 3 & \longrightarrow & x := 0 \end{array}$$

[1]We note that finiteness of processes, variables, and domains is due the fact that our approach in this paper is based on model checking, which is most effective in the context of finite models.

A *global state* s of a distributed system is an instance of the local state of its processes. We denote the set of all states of a distributed system $\mathcal{G}$ by S (called its *state space*). The concurrent execution of the set of all local algorithms defines a *distributed algorithm*. We say that an action of a process p is *enabled* in a state s if and only if its guard is true in s. By extension, process p is said enabled in s if and only if at least one of its actions is enabled in s. An action can be executed only if its guard is enabled. If *atomic* execution of an action in state s results in state s', we call (s, s') a *transition*.

Definition 1 (Computation): A *computation* of a distributed system is a maximal sequence of states $\overline{\sigma} = s_0 s_1 \ldots$, such that for all $i \geq 0$, each pair (s_i, s_{i+1}) is a transition. Maximality of a computation means that the computation is either infinite or eventually reaches a terminal state; *i.e.*, a state where no action is enabled. ■

B. Probabilistic Model Checking

In order to reason about distributed systems, we focus on their state space and set of transitions. Let AP be a set of atomic propositions.

Definition 2 (Markov Chain): A *discrete-time Markov chain (DTMC)* is a tuple $D = (S, S^0, \mathbb{P}, L)$, where

- S is the finite state space
- $S^0 \subseteq S$ is the set of initial states
- $\mathbb{P} : S \times S \to [0, 1]$ is a function such that for all $s \in S$, we have
$$\sum_{s' \in S} \mathbb{P}(s, s') = 1$$
- $L : S \to 2^{AP}$ is a labeling function assigning to each state a set of atomic propositions. ■

It is straightforward to see that one can represent a distributed algorithm as a Markov chain. In a Markov chain, the fact that $\mathbb{P}(s, s') \neq 0$ for two states $s, s' \in S$ stipulates there is a transition from s to s' that can be executed with some probability. We emphasize that representing a distributed algorithm in our framework based on a Markov chain does not make our approach in this paper suitable for only probabilistic distributed algorithms (e.g., [15]). In particular, if a distributed algorithm is not probabilistic, then it can be modeled as a Markov chain, where the probability of all outgoing transitions from each state are equal. Thus, the definition of computation in a Markov chain is identical to the one presented in Subsection II-A.

Example: The Markov chain of the example presented in Subsection II-A is shown in Figure 1. Each state is labeled by the value of variable x. Each transition is annotated by the probability of its occurrence. In particular,

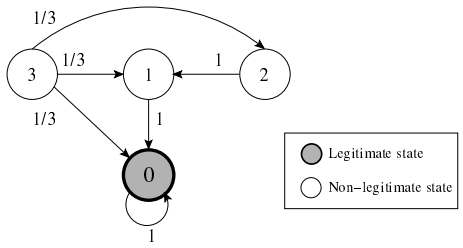


Figure 1. Markov chain of the guarded commands in Subsection II-A.

the probability of outgoing transitions from states 0, 1, and 2 are 1, as these transitions are the only outgoing transitions. All outgoing transitions from state 3 have probability $\frac{1}{3}$ since the corresponding guarded command program is non-deterministic but also non-probabilistic. Hence, these transitions should execute with the same probability. Of course, such non-determinism also depends on the scheduler. We will explain the role of scheduler in Section IV.

Probabilistic model checking is based on the definition of a probability measure over the set of paths that satisfy a given property specification. Our specification language in this paper is the Probabilistic Computation Tree Logic (PCTL). This logic is especially designed for reasoning about reliability.

*Definition 3 (*PCTL *Syntax):* Formulas in PCTL [14] are inductively defined as follows:

$$\varphi ::= p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \mathbb{P}_{\sim\lambda}(\varphi_1 \,\mathcal{U}^h\, \varphi_2)$$

where $p \in AP$, $\sim \in \{<, \leq, \geq, >\}$ is a comparison operator, and λ is probability threshold. The sub-formula $\varphi_1 \,\mathcal{U}^h\, \varphi_2$ is the classic (bounded/unbounded) "until" operator. ■

*Definition 4 (*PCTL *Semantics):* Let $\overline{\sigma} = s_0 s_1 \ldots$ be an infinite computation, i be a non-negative integer, and $\models$ denote the *satisfaction* relation. Semantics of PCTL is defined inductively as follows:

$$\begin{array}{lll}
\overline{\sigma}, i \models \mathit{true} & & \\
\overline{\sigma}, i \models p & \text{iff} & p \in L(s_i) \\
\overline{\sigma}, i \models \neg\varphi & \text{iff} & \overline{\sigma}, i \not\models \varphi \\
\overline{\sigma}, i \models \varphi_1 \vee \varphi_2 & \text{iff} & (\overline{\sigma}, i \models \varphi_1) \ \vee \ (\overline{\sigma}, i \models \varphi_2) \\
\overline{\sigma}, i \models \varphi_1 \,\mathcal{U}^h\, \varphi_2 & \text{iff} & \\
\multicolumn{3}{l}{\qquad \exists k \geq i : (k - i = h) \wedge (\overline{\sigma}, k \models \varphi_2) \wedge} \\
\multicolumn{3}{l}{\qquad\qquad \forall j : i \leq j < k : \overline{\sigma}, j \models \varphi_1.}
\end{array}$$

In addition, $\overline{\sigma} \models \varphi$ holds iff $\overline{\sigma}, 0 \models \varphi$ holds. Finally, for a state s, we have $s \models \mathbb{P}_{\sim\lambda}\varphi$ iff the probability of taking a path from s that satisfies φ is $\sim \lambda$. ■

Following Definition 4, we use the usual abbreviation $\Diamond\varphi \equiv (\mathit{true}\,\mathcal{U}\,\varphi)$ for "eventually" formulas. Moreover, $\Box\varphi \equiv \neg\Diamond\neg\varphi$ is the classical "globally" path formula.

Example: It is straightforward to see that the Markov chain in Figure 1 satisfies the following PCTL properties:

- $\mathbb{P}_{\leq \frac{1}{3}}(x = 3) \Rightarrow \Diamond^3(x = 0)$
- $\mathbb{P}_{\geq 1}\Diamond(x = 0)$, which is logically equal to $\mathbb{P}_{\geq 1}(1 \leq x \leq 3)\,\mathcal{U}\,(x = 0)$

Definition 5: Let $D = (S, S^0, \mathbb{P}, L)$ be a Markov chain. A *state predicate* is a subset of S. ■

Observe that a state predicate is a PCTL formula constructed by only atomic propositions and Boolean operators $\neg$ and $\vee$.

C. Self-stabilization

Intuitively, a self-stabilizing system is one that if its execution starts from any *arbitrary* state, then it always reaches a good behavior within a finite number of steps (called the *convergence* property) and after convergence, it behaves normally unless its state is perturbed by transient faults (called *closure* property). The so called "good behavior" is normally modeled by a state predicate called *legitimate states*. Since a self-stabilizing system can start executing from any arbitrary state, in the context of Markov chains, we assume that $S = S^0$; *i.e.*, an initial state can be any state in the state space. We formally define the notion of legitimate states using PCTL as follows.

Definition 6 (Self-stabilization): Let $D = (S, S^0, \mathbb{P}, L)$ be a Markov chain, representing a distributed algorithm, and LS be a state predicate. We say D is *self-stabilizing* for *legitimate states* LS iff the following two conditions hold:

- *(Closure)* For each state $s \in LS$, if there exists a state s' such that $\mathbb{P}(s, s') \neq 0$, then $s' \in LS$; *i.e.*, execution of a transition in LS results in a state in LS.
- *(Convergence)* We have $\mathbb{P}_{\geq 1}\Box\Diamond LS$; *i.e.*, starting from any arbitrary state, the probability of reaching the legitimate states is always 1, and D has no cycles in $\neg LS$. ■

In Definition 6, the convergence property is also called *recovery* and a computation that starts from a state in $\neg LS$ and reaches a state in LS is called a *recovery path*.

Example: It is straightforward to see that the Markov chain in Figure 1 is self-stabilizing for $LS \equiv (x = 0)$.

III. Evaluation Method Based on Probabilistic Model Checking

In this section, we describe our method for evaluating the performance of self-stabilizing algorithms. First, in Subsection III-A, we describe the shortcomings of the conventional approaches to characterize the performance of self-stabilizing algorithms. Then, Subsection III-B presents an intuitive description of our approach, while Subsection III-C elaborates on our evaluation technique in formal terms.

A. Shortcomings of Conventional Approaches

The conventional metric to evaluate the performance of a self-stabilizing algorithm is asymptotic computational complexity in terms of rounds, cycles, or recovery time:

- A *round* [11] is a shortest computation fragment in which each process executes at least one step.
- A *cycle* is a shortest computation fragment in which each process executes at least one complete iteration of its repeatedly executed list of commands.
- *Recovery time* (also called *stabilization time*) is the number of global actions that has to be executed from an arbitrary state to a legitimate state.

Thus, one can characterize the performance of a self-stabilizing algorithm as the worst case number of rounds in terms of the big $\mathcal{O}$ notation of a parameter of the distributed system. For example, if the topology of a distributed system is a tree, then the performance could be $\mathcal{O}(h^2)$ rounds, where h is the height of the tree. We argue that such characterization of performance is too abstract and does not reflect all the realities of deploying a distributed algorithm in practice. For example, the asymptotic computational complexity abstracts away the following parameters that can be potentially crucial in practice:

- The constants removed in computing $\mathcal{O}$ can be large enough to affect the performance of the algorithm.
- The smaller polynomials removed in computing $\mathcal{O}$ can be large enough to have significant impact on the performance of the algorithm.
- The worst case complexity may rarely occur.

We believe these abstractions make the asymptotic computational complexity a less attractive metric to analyze the performance of self-stabilizing algorithms.

B. Sketch of Our Approach

In order to address the aforementioned issues, we advocate computing the average case performance in terms of recovery time by taking into account the characteristics of each state outside the set of legitimate states of a self-stabilizing algorithm. To this end, we utilize state exploration and, hence, model checking. In particular, given a self-stabilizing algorithm, we take the following steps:

1) We compute the probability of reachability of legitimate states for each state outside legitimate states within h number of steps.
2) Then, we compute the expected mean value of recovery time for each state outside the set of legitimates states.
3) Finally, we calculate the expected mean value of recovery time for the set of non-legitimate states.

It is straightforward to see that the result of the last step characterizes how fast a self-stabilizing algorithm recovers.

Moreover, this metric encodes average case analysis, which is a more fair measure for performance analysis. One can use this metric to compare the performance of two self-stabilizing algorithms. The remainder of this section describes how we employ probabilistic model checking techniques to rigorously compute the expected mean value of recovery time.

C. Detailed Description

Let $D = (S, S^0, \mathbb{P}, L)$ be a Markov chain with legitimate states LS and $\mathfrak{s}$ be state in $\neg LS$. Following Definition 4, the PCTL property

$$\mathbb{P}_{\geq p}(\mathfrak{s} \Rightarrow \Diamond^h LS)$$

holds in D, if starting from $\mathfrak{s}$ the probability of reaching LS in h steps is greater than or equal to p.

In order to analyze the speed of recovery of a self-stabilizing algorithm, we consider the other side of the coin by computing the probability of truthfulness of the following expression

$$(\mathfrak{s} \Rightarrow \Diamond^h LS)$$

for each state $\mathfrak{s} \in \neg LS$. Let μ be the probability distribution on recovery paths and $R(\mathfrak{s})$ denote the length of recovery path that starts from $\mathfrak{s}$. One can compute the probability of existence of recovery paths of length at most N from state $\mathfrak{s}$ as follows:

$$\mathbb{P}\{R(\mathfrak{s}) \leq N\} = \sum \big\{\mu(\overline{\sigma} = s_0 s_1 \dots) \mid (s_0 = \mathfrak{s}) \ \wedge \ (\overline{\sigma} \models \Diamond^h LS) \ \wedge \ (h \leq N)\big\} \quad (1)$$

Example: For the Markov chain in Figure 1, the probability of recovery of length at most 2 from state 3 to state 0 is: $1/3 + 1/3 = 2/3$.

One can also compute the expected mean value of $R(\mathfrak{s})$, that is, the average length of all recovery paths that start from state $\mathfrak{s}$. Again, this value can be computed by direct summation as follows:

$$\mathbb{E}\{R(\mathfrak{s})\} = \sum \big\{\mu(\overline{\sigma} = s_0 s_1 \dots) \times h \mid (s_0 = \mathfrak{s}) \ \wedge \ (\overline{\sigma} \models \Diamond^h LS)\big\} \quad (2)$$

Example: For the Markov chain in Figure 1, the expected mean value of recovery steps for state 3 is

$$1 \times \frac{1}{3} + 2 \times \frac{1}{3} + 3 \times \frac{1}{3} = 2$$

Observe that unlike the conventional characterization of self-stabilizing systems, where the system can start executing from all initial states with the same probability (or equally, faults can perturb the system to any state with the same probability), we argue that in practice this is not the case. Thus, we associate a probability $p(\mathfrak{s})$ to each state $\mathfrak{s} \in \neg LS$, such that

$$\sum_{\mathfrak{s} \in \neg LS} p(\mathfrak{s}) = 1$$

Subsequently, the mean expected value of recovery steps for a self-stabilizing algorithm represented by a Markov chain D is the following:

$$\mathbb{E}\{R(D)\} = \sum \big\{\mathbb{E}\{R(\mathfrak{s})\} \times p(\mathfrak{s}) \mid \mathfrak{s} \in \neg LS\big\} \quad (3)$$

Unlike classic asymptotic complexity analysis, Equation 3 explicitly takes into account several parameters that are of importance in both theory and practice for analyzing the efficiency of self-stabilization. These factors include the likelihood of arbitrary initializations and the number of states in $\neg LS$, from where recovery is fast (respectively, is slow).

Example: For the Markov chain in Figure 1, assuming equal probability for all non-legitimate states, the expected mean value of recovery is

$$1 \times \frac{1}{3} + 2 \times \frac{1}{3} + 2 \times \frac{1}{3} = 1.66$$

It is easy to see that if faults with non-uniform probability distribution perturb the state of the system, the expected mean value of recovery steps can dramatically change. For instance, if the fault that reaches state 3 occurs with probability 90% and states 2 and 1 are reached by probability 5%, then the expected mean value of recovery steps is

$$1 \times \frac{5}{100} + 2 \times \frac{5}{100} + 2 \times \frac{90}{100} = 1.95$$

Our performance metric can be, henceforth, applied to compare the performance of two self-stabilizing algorithms (*i.e.,* by comparing their expected mean value of recovery steps). Formally:

> Given $D_1 = (S_0, S_1^0, \mathbb{P}_1, L_1)$ and $D_2 = (S_2, S_2^0, \mathbb{P}_2, L_2)$, we say that D_1 outperforms D_2 iff
>
> $$\mathbb{E}\{R(D_1)\} \ < \ \mathbb{E}\{R(D_2)\}$$

IV. Other Benefits of our Approach

In this section, we elaborate on the benefits of using our metric to determine the performance self-stabilizing algorithms in addition to rigorous average case analysis. We classify these benefits in terms of the parameters that our metric can take into account.

Impact of scheduler: Formally, a *scheduler* (also called a *daemon*) is a predicate that defines a set of admissible computations. In our formulation, we did not consider a scheduler, but there exist several types of schedulers [12]. The most liberal scheduler, is a *distributed unfair* scheduler, where 'distributed' means that any subset of enabled processes may be scheduled for execution at any given time and 'unfair' means that if a process p is continuously enabled, then p may never be chosen by the scheduler unless p is the only enabled process. A *central*

scheduler is one that executes only one action among all enabled actions; i.e., actions of processes are executed in an interleaving fashion. A strongly *fair* scheduler eventually chooses an action that is continuously enabled. A weakly fair scheduler eventually chooses an action of a process p, if p is continuously enabled. Observe that in practice, a central scheduler resembles the scheduler of a uni-processor system and a distributed scheduler embodies the scheduler of a multi-processor or multi-core system. Moreover, a fair scheduler is one that implements some aging algorithm. All these types of schedulers may rule out some computations, which, in turn, may change the mean expected value of recovery time of an algorithm. In order to incorporate the impact of a scheduler on performance analysis, one can generate the Markov chain of a set of guarded commands by enforcing scheduling predicates. For instance, if in a state, several actions are enabled, then a distributed scheduler may potentially execute any combination of the enabled actions. Such combinations can be captured in the resulting Markov chain in a straightforward fashion. We note that the scheduler itself can also be probabilistic. For instance, a scheduler of a uni-processor or multi-core system may enforce certain policies by assigning non-uniform probability of execution of actions (e.g., the scheduler gives higher priority to recovery actions). Such cases can elegantly be woven into our framework.

Likelihood of occurrence of faults or initialization: A serious drawback of asymptotic performance analysis is that it completely abstracts away the fact that arbitrary initializations and faults often do not occur with uniform probability in practice. In fact, non-uniform probability distribution of faults, may significantly change the performance of a self-stabilizing algorithm. Observe that Equation 3, addresses this shortcoming of asymptotic performance evaluation. This means that if faults that reach states from where recovery time is longer are not probable, then the algorithm is more likely to perform much better than the worst case big $\mathcal{O}$ complexity.

Insights on impact of adversaries: Since recovery time may differ from different non-legitimate states, faults that perturb the system to states from where recovery is slow are arguably more severe. In other words, severe faults, whose occurrence will require longer recovery, will degrade the performance of the system, as the system may spend more time on recovery than normal execution. Thus, if an adversary intends to attack the system, it can cause greater damage by injecting faults that are more severe. Since our method can effectively identify such faults, one can obtain useful insights about the system and revise the system, so that it is more resilient to more severe faults (e.g., using state encoding and memory redundancy techniques, see Section VI).

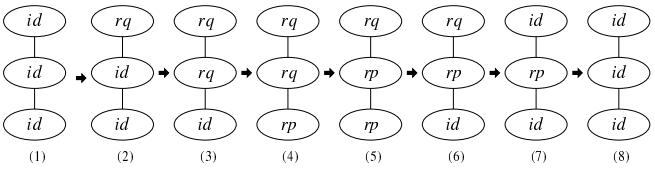


Figure 2. An execution scenario of PIF

Blindness to network topology and communication: Another benefit of our approach is its generality with respect to the underlying network topology and communication technology. Notice that our approach can be applied on any self-stabilizing algorithm non-withstanding the input language (be it guarded commands, I/O automata [17], or process algebras [2]), type of communication (be it shared memory or message passing), or network topology (be it a ring, a tree, or an arbitrary graph). Regardless of the presentation of an algorithm, we transform the algorithm into a Markov chain for our analyses. Having said that, it can be the case that an algorithm exhibits a different performance for similar topologies with small differences (e.g., a tree and a chain of processes). These details cannot be analyzed easily using abstract metrics such as the big $\mathcal{O}$ notation.

Parametric analysis: Although classic model checkers take a concrete model as input, there have recently been advances on parametric verification. For instance, using parametric model checking, one can analyze an algorithm for arbitrary size trees. Using such methods will allow us to analyze the performance of algorithms for more general classes of network topologies.

In Section V, we will present experimental evidence that touches upon some of the above benefits.

V. Implementation and Experimental Results

In this section, we present our case study on distributed Propagation of Information with Feedback (PIF) in rooted trees. Figure 2 shows a simple execution scenario of an algorithm for PIF for three processes arranged on a line. Each process has three states: idle, request, and response (denoted $\{id, rq, rp\}$). In this scenario, initially, all processes are idle (step 1). Then, the root makes a request (step 2). This request propagates all the way to the leaf, where a response is generated (steps 3-4). A process whose all children are responding also responds, except for root, which becomes idle (steps 5 and 7). A responding process becomes idle when its parent is responding (steps 6 and 8).

We focus on three non-probabilistic self-stabilizing algorithms. These algorithms achieve PIF given any arbitrary state. The first algorithm [4] is snap-stabilizing with performance of $\mathcal{O}(h^2)$ rounds, where h is the height of the rooted tree of the underlying network. In the sequel, we refer to this algorithm as Snap1. The second algorithm [4] is snap-stabilizing with performance of $\mathcal{O}(1)$ rounds. In the sequel,

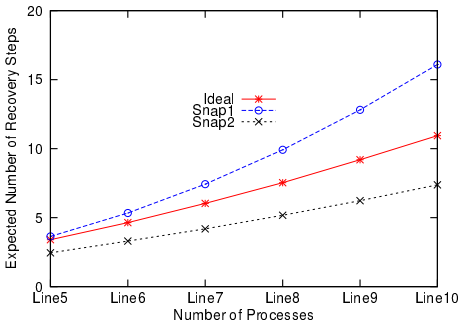


Figure 3. Mean recovery time for three PIF algorithms

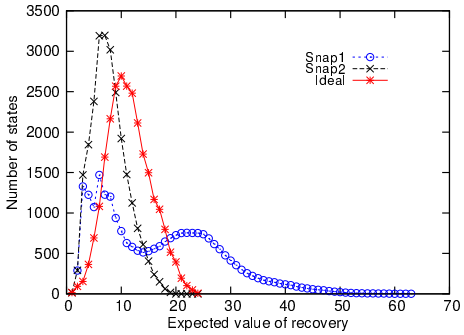


Figure 4. Recovery time skewness of PIF for line 10

we refer to this algorithm as Snap2. The third algorithm [20] is an ideal-stabilizing algorithm with performance $\mathcal{O}(h^2)$. We refer to this algorithm as Ideal. In all algorithms, the set of states reachable in the absence of faults from the state where all processes are in state id identify legitimate states.

A. Implementation

The three aforementioned algorithms are modeled in the probabilistic model checker PRISM [16] as discrete-time Markov chains. We use the *Reward* mechanism of PRISM to record and reason about the recovery time from each state. Specifically, the reward "$\neg LS : steps + 1$", where initially $steps = 0$, specifies that in a computation, when a state in $\neg LS$ is reached, the recovery path has an additional step. Thus, the expected mean value of recovery time can be computed using the value of $steps$. PRISM computes the expected value of $steps$ only for a single initial state (*i.e.*, Equation 2 in Section III). However, using the *Filter* mechanism in PRISM, we can compute the expected value of $steps$ for a set of initial states, namely, all states in LS (*i.e.*, Equation 3 in Section III). Likewise, using filters, one can compute the expected mean value of recovery time, from certain set of states as well. Examples include, set of states reached by certain types of faults or faults that occur in certain processes.

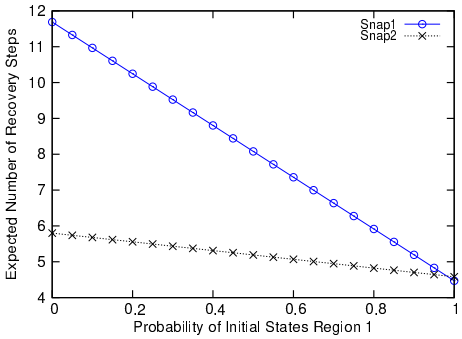
B. Experimental Results

This subsection is organized in a top-down fashion: we first present our high-level results on comparing the performance of algorithms and then describe insights obtained by detailed analysis of the algorithms. We will employ our insights gained in this section to design a state encoding scheme that improves the performance of an algorithm in Section VI.

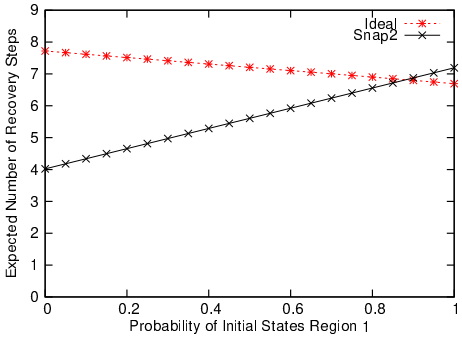
1) Expected mean recovery time: Figure 3 shows the expected mean value of recovery time of the three PIF algorithms for different number of processes. As can be seen, for line topology of lengths 5-10, algorithm Snap2 exhibits faster recovery time than Ideal and Snap1. However, algorithm Snap1's performance seems to diverge from the other two with a faster pace. This is due to the fact that in Snap1, as the number of processes increase, the number of states from where recovery is slow grows more rapidly than the other two algorithms. Also, observe that although Snap1 and Ideal have identical asymptotic performance, Ideal outperforms Snap1 and it does not diverge as fast as Snap1 as compared to Snap2.

2) Recovery time distribution: As mentioned above, the distribution of recovery time over states determines the overall recovery time of an algorithm. To analyze the effect of this distribution, consider Figure 4, which shows the number of states in $\neg LS$ in terms of recovery time distribution of PIF for line 10. The skewness value of a graph can represent the performance of the algorithm. If the majority of states are concentrated on lower values of recovery, then the expected mean value of recovery time decreases. Moreover, a larger absolute value implies a better performance for the algorithm. That is, recovery for a larger number of states is faster. As we can observe, the skewness of Snap2 is towards the left, which causes the expected mean value of recovery time to be less than the other algorithms. The same interpretation can be applied for comparing Ideal and Snap1. In particular, skewness values for Snap2, Ideal, and Snap1 are 2.26, 1.86, and 0.88, respectively. Hence, Snap2 has the best performance.

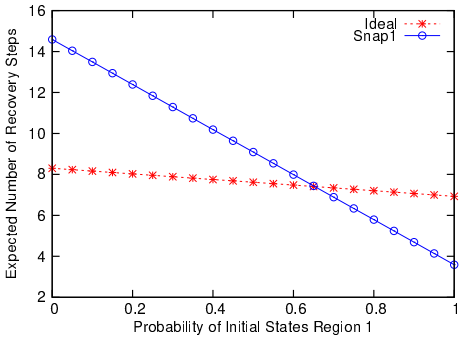
3) Impact of fault probability distribution: In practice, faults occur with a non-uniform probability distribution. As described in Equation 3, this probability distribution can significantly affect the performance of a self-stabilizing algorithm. Our claim is that since there exist states from where algorithms Snap1 and Ideal show faster recovery than Snap2, if some faults reach a set of states with higher probability, then the expected recovery time may change significantly. Figure 5 validates our claim (Region 1 refers to a state predicate in $\neg LS$, from where Snap1 performs better than Snap2). As can be seen, in Figure 5(a) (respectively, Figure 5(b)), there are cases, where Snap2 underperforms Snap1 (respectively, Ideal). This situation is more clear in case of Ideal and Snap1 (see Figure 5(c)); *i.e.*, there are more cases where Snap1 outperforms Ideal,



(a) Performance of Snap1 vs. Snap2



(b) Performance of Ideal vs. Snap2



(c) Performance of Ideal vs. Snap1

Figure 5. Impact of fault probability distribution (line 8)

although Ideal has better uniform performance than Snap1 (recall Figure 3). We note that since predicate $\neg LS$ for Snap1 and Snap2 are identical and differ from $\neg LS$ of Ideal, we cannot compare them in the same graph.

4) Impact of type of and location of faults: Studying the impact of place of occurrence and type of faults is a part of sensitivity analysis, which is a crucial step in deploying distributed systems in practice. Figure 6 compares the overall expected recovery time with the recovery time when certain faults with respect to type and location occur, for all three algorithms for line of size 12. For instance, if the state of the root process becomes rq in $\neg LS$, then in all three algorithms, the recovery time from these states is less than the overall expected recovery time of the algorithms. This is

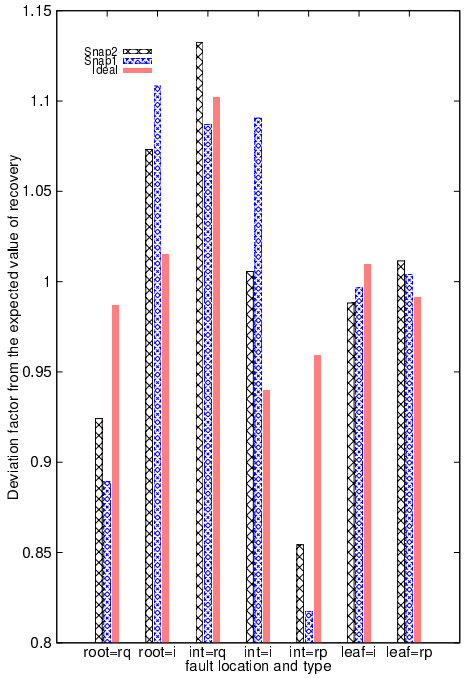


Figure 6. Comparison of recovery time based on the place and type of of faults (line 12)

also the case when the state of an intermediate (*i.e.*, non-root and non-leaf) process becomes rp. On the contrary, faults that change the state of intermediate processes to rq are more severe. This is of course expected, since such faults initiate two waves of message propagation for stabilization. This sensitivity analysis provides us with useful insight to implement more efficient protocols. One approach that benefits from sensitivity analysis is *state encoding* described in the next section.

VI. Employing State Encoding to Improve Performance

State encoding consists in assigning bit patterns to abstract algorithm states. For example, two states s and s' can be associated to bits `0` and `1`, respectively. Another possibility is to have bit patterns of size two and mapping `00` to s and `01`, `10`, and `11` to s'. In the first case, there is a bijection between bit pattern and abstract states. In the second case, the mapping is injective, but not surjective. The purpose of such state encoding is twofold:

1) increase the proportion of LS as compared to $\neg LS$ states by making states appearing in $\neg LS$ less likely to appear (compared to the default choice of a bijective mapping) if bits can, for instance, randomly get flipped.
2) decrease the average expected recovery time by making states that occur in long recovery time executions less likely to appear than those belong to short recovery time executions.

	id				rq				rp			
	size	$\mathbb{E}(R)$	Encoding	$\mathbb{E}'(R)$	**size**	$\mathbb{E}(R)$	Encoding	$\mathbb{E}'(R)$	**size**	$\mathbb{E}(R)$	Encoding	$\mathbb{E}'(R)$
root	10	1.9	$\{00\}$	1.62	4	1.5	$\{01, 10, 11\}$	1.4	0	–	–	–
Process 2	4	1.5	$\{001, 010, 011\}$	1.36	6	2.2	$\{000\}$	4.94	4	1.5	$\{100, 101, 110, 111\}$	1.15
Process 3	2	2.5	$\{001\}$	1.35	10	1.75	$\{000\}$	1.42	2	1.25	$\{010, 011, 100, 101, 110, 111\}$	2.10
Leaf	7	1.2	$\{00\}$	1.79	0	–	–	–	7	1.6	$\{01, 11, 10\}$	1.26

Table I
STATE ENCODING FOR PIF (LINE OF 4)

Our claim in this section is that the insights gained through our performance analysis technique, make it possible to design encodings that can improve the performance of an algorithm. Note that state encoding is only an *implementation* technique and does not change the behavior of the original algorithm.

Consider a line of four processes for Snap1 algorithm. We make the following observations (summarized in Table I):

- In case of the root process both states id or rq appear in $\neg LS$, but id appears 2.5 times more. Moreover, the expected recovery time when the state of root is id is 1.9, while the expected recovery time when the state of root is rq is 1.5. Thus, assigning more bit patterns to rq will results in a smaller-size $\neg LS$ as well as decreasing the expected recovery time. A 2-bit state mapping is the following: $\{00\} \mapsto id$ and $\{01, 10, 11\} \mapsto rq$. This encoding will result in new recovery time $\mathbb{E}' = 1.62$ for state id and $\mathbb{E}' = 1.4$ for state rq.
- The state of the second process can be id, rq, or rp and all three appear in $\neg LS$. However, state rq appears in 6 global states, while the other two appear in 4 global states. Also, the expected recovery time for states id and rp is 1.5, while the expected recovery time for state rq is 2.2. Thus, reducing the occurrences of state rq should decrease the expected recovery time. Two bits are necessary to encode three states, but we should not map two different bit patterns to state rq. Thus, we use a 3-bit encoding to decrease the proportion of non-legitimate states as follows: $\{000\} \mapsto rq$, $\{001, 010, 011\} \mapsto id$, and $\{100, 101, 110, 111\} \mapsto rp$. Notice that the choice for the state mapping of states id and rp is arbitrary.
- In case of the third process, based on the recovery time and number of states shown in Table I, allocating more bit patterns to state id would have two opposite effects: reducing the proportion of non-legitimate states, but increasing the average expected recovery time. On the contrary, allocating more bit patterns to state rp decreases both the proportion of non-legitimate states and the expected recovery time. Based on these observations, we incorporate the state encoding shown in Table I.
- The state of the leaf process can be either id or rp. Based on the number of their appearance in recovery time, we incorporate the state encoding shown in Table I.

States	$\mathbb{E}$	$\mathbb{E}'$	# of Encoded States
(id, id, rq, id)	2.0	2.66	3
(id, id, rq, rp)	1.5	1.33	9
(id, rq, id, id)	2.62	4.19	1
(id, rq, id, rp)	2.31	5.20	3
(id, rq, rq, id)	3.25	3.20	1
(id, rq, rq, rp)	2.5	2.20	3
(id, rq, rp, id)	1.0	1	6
(id, rq, rp, rp)	1.5	1.2	18
(id, rp, rq, id)	1.5	1.99	4
(id, rp, rq, rp)	1.0	1	12
(rq, id, rq, id)	1.5	1.99	9
(rq, id, rq, rp)	1.0	1	27
(rq, rp, rq, id)	2.0	2.36	12
(rq, rp, rq, rp)	1.5	1.14	36

Table II
IMPACT OF ENCODING ON PIF RECOVERY TIME (LINE OF 4)

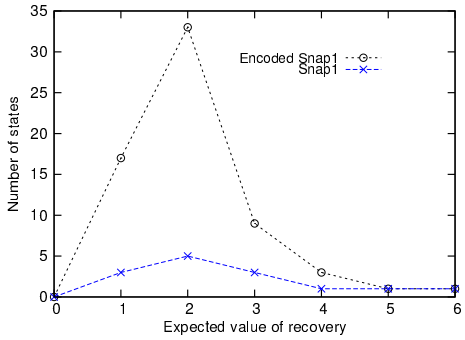


Figure 7. Skewness of Snap1 before and after encoding (line 4)

The expected recovery time for each process state after encoding is denoted by $\mathbb{E}'$ in Table I. Observe that for some processes, encoding makes recovery slower, but the technique is globally very effective. To clarify this, consider the recovery time before and after encoding for each global state in Table II. By taking into account that each global state may be replicated, the overall recovery time for algorithm Snap1 before encoding is $\mathbb{E} = 2.36$ steps, while after encoding is $\mathbb{E}' = 1.47$ steps. Figure 7 also shows the skewness graph of Snap1 before (0.81) and after (1.56) encoding.

VII. CONCLUSION

In this paper, we focused on automated performance analysis of distributed self-stabilizing algorithms using probabilistic state exploration techniques. We argue that the commonly used asymptotic complexity metric – the big $\mathcal{O}$ no-

tation – abstracts away many factors crucial to performance analysis. Moreover, it cannot take into account real-world constraints such as the likelihood of occurrence of faults. We proposed a new metric, namely, the mean expected value of number of global steps to complete stabilization. This metric can be elegantly measured using off-the-shelf probabilistic model checkers. We supported our claims by conducting rigorous experiments on three self-stabilizing algorithms for distributed Propagation of Information with Feedback (PIF) in rooted trees. We also showed that the insights gained by our experiments led to more efficient implementations using state encoding.

As for future work, there is a diverse set of new research directions. For instance, one can design parametric model checking techniques to analyze the performance of algorithms when the exact number of processes is not given. Automated state encoding based on state exploration is another interesting problem. Yet another challenging problem is to devise probabilistic synthesis techniques that can revise an existing protocol to improve its performance.

VIII. Acknowledgements

This research was supported in part by Canada NSERC Discovery Grant 418396-2012 and NSERC Strategic Grant 430575-2012.

References

[1] J. Adamek, M. Nesterenko, and S. Tixeuil. Using abstract simulation for performance evaluation of stabilizing algorithms: The case of propagation of information with feedback. In *Proceedings of 14th International Conference on Stabilization, Safety, and Security in Distributed Systems (SSS)*, pages 126–132, 2012.

[2] M. Alexander and W. Gardner. *Process Algebra for Parallel and Distributed Processing*. Chapman & Hall/CRC, 2008.

[3] L. Blin, M. Gradinariu Potop-Butucaru, S. Rovedakis, and Sébastien Tixeuil. A new self-stabilizing minimum spanning tree construction with loop-free property. In *Proceedings of the International Conference on Distributed Computing (DISC 2009)*, September 2009.

[4] A. Bui, A. K. Datta, F. Petit, and V. Villain. Snap-stabilization and pif in tree networks. *Distributed Computing*, 20(1):3–19, 2007.

[5] E. Caron, F. Desprez, F. Petit, and C. Tedeschi. Snap-stabilizing prefix tree for peer-to-peer systems. *Parallel Processing Letters*, 20(1):15–30, 2010.

[6] V. Chernoy, M. Shalom, and S. Zaks. A self-stabilizing algorithm with tight bounds for mutual exclusion on a ring. In *Proceedings of 22nd International conference on Distributed Computing (DISC)*, pages 63–77, 2008.

[7] A. K. Datta, L. L. Larmore, and P. Vemula. Self-stabilizing leader election in optimal space under an arbitrary scheduler. *Theor. Comput. Sci.*, 412(40):5541–5561, 2011.

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

[9] S. Dolev. *Self-stabilization*. MIT Press, March 2000.

[10] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago J. Theor. Comput. Sci.*, 1997, 1997.

[11] S. Dolev, A. Israeli, and S. Moran. Uniform dynamic self-stabilizing leader election. *IEEE Transactions on Parallel Distributed Systems*, 8(4):424–440, 1997.

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

[13] M. Flatebo and A. K. Datta. Simulation of self-stabilizing algorithms in distributed systems. In *Annual Simulation Symposium*, pages 32–41, 1992.

[14] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspect of Computing (FAOC)*, 6(5):512–535, 1994.

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

[16] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Computer Aided Verification (CAV)*, pages 585–591, 2011.

[17] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[18] N. Mitton, B. Séricola, S. Tixeuil, E. Fleury, and I. Guérin-Lassous. Self-stabilization in self-organized multihop wireless networks. *Ad Hoc and Sensor Wireless Networks*, 11(1-2):1–34, January 2011.

[19] N. Mullner, A. Dhama, and O. Theel. Derivation of fault tolerance measures of self-stabilizing algorithms by simulation. In *Simulation Symposium, 2008. ANSS 2008. 41st Annual*, pages 183 –192, april 2008.

[20] M. Nesterenko and S. Tixeuil. Ideal stabilization. In *Proceedings of the 25th IEEE International Conference on Advanced Information Networking and Applications (AINA)*, pages 224–231, 2011.

[21] F. Ooshita and S. Tixeuil. In *Proceedings of the International Conference on Stabilization, Safety, and Security in Distributed Systems (SSS 2012)*, pages 49–63, 2012.

[22] S. Tixeuil. *Algorithms and Theory of Computation Handbook, Second Edition*, chapter Self-stabilizing Algorithms, pages 26.1–26.45. Chapman & Hall/CRC Applied Algorithms and Data Structures. CRC Press, Taylor & Francis Group, November 2009.

[23] M. Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 327–338, 1985.

[24] G. Varghese and M. Jayaram. The fault span of crash failures. *J. ACM*, 47(2):244–293, 2000.