Derivation of Fault Tolerance Measures of Self-Stabilizing Algorithms by Simulation[*]

Nils Müllner, Abhishek Dhama, Oliver Theel
Carl von Ossietzky University of Oldenburg
Department of Computer Science
D-26111 Oldenburg, Germany
Email: oliver.theel@uni-oldenburg.de

Abstract

Fault tolerance measures can be used to distinguish between different self-stabilizing solutions to the same problem. However, derivation of these measures via analysis suffers from limitations with respect to scalability of and applicability to a wide class of self-stabilizing distributed algorithms. We describe a simulation framework to derive fault tolerance measures for self-stabilizing algorithms which can deal with the complete class of self-stabilizing algorithms. We show the advantages of the simulation framework in contrast to the analytical approach not only by means of accuracy of results, range of applicable scenarios and performance, but also for investigation of the influence of schedulers on a meta level and the possibility to simulate large scale systems featuring dynamic fault probabilities.

Keywords: *Fault Tolerance, Self-Stabilization, Simulation, Reliability, Availability*

1 Introduction

High dependability has become an important design requirement for distributed systems owing to their pervasive usage in safety-critical applications. System designers endow such mission-critical systems with *fault tolerance* to meet the dependability requirements. A fault-tolerant distributed system strives to fulfill its specifications in spite of failures.

Fault-tolerant distributed systems can be broadly classified into two categories: *masking fault-tolerant systems* and *non-masking fault-tolerant systems* [14]. Masking fault-tolerant distributed systems are able to "disguise" the faults and function according to the specifications as long as the faults belong to particular classes. *Au contraire*, an external observer is able to witness *incorrect* behavior for a certain period of time before non-masking fault-tolerant systems again function according to their specification.

Self-stabilization is an elegant technique to provide non-masking fault tolerance. A self-stabilizing system – from any initial state – reaches a legal set of states within a finite period of time and does not leave this set of states on its own [9]. The self-stabilization property becomes extremely attractive under the assumption that the system is afflicted only by *transient* faults (*i.e.,* non-permanent and non-intermittent faults) that only perturb the system state. Thus, a self-stabilizing system recovers from "bursts" of transient faults within finite time irrespective of the quantum of perturbation. Due to this notion of autonomy, self-stabilizing systems provide many algorithmic solutions applied in the context of, among others, sensor networks, wireless networks and ad hoc networks [12, 17].

A system designer, when confronted with the dilemma of choosing amongst multiple fault-tolerant solutions to the same problem, uses *fault tolerance measures* to arrive at a decision. Among the fault tolerance measures defined in the literature, *reliability*, *instantaneous availability* and *limiting availability* are most commonly used [20]. Due to almost three decades of sustained research, the literature is replete with multiple self-stabilizing algorithms to solve many problems in distributed computing. Usually the quality of a self-stabilizing algorithm is characterized by the worst-case time/space complexities. However, these measures might not really capture the behavior of a system in an implementation scenario as such extreme conditions manifest very rarely during a system's lifetime. In addition, two self-stabilizing algorithms might have the same worst-case complexity while their average-case complexity might differ.

To overcome these limitations, notions of reliability, instantaneous availability and limiting availability have been defined for self-stabilizing algorithms in [7] and a method to determine these measures for a sub-class of *silent* self-stabilizing systems is outlined (see Section 2.2 for a brief summary). In [7] fault tolerance measures for non-masking fault-tolerant systems are defined for the first time. Yet, the analysis procedure presented is limited with respect to scal-

[*] This work was partly supported by the German Research Foundation (DFG) under grants GRK 1076/1 "TrustSoft" and SFB/TR 14 "AVACS."

1080-241X/08 $25.00 © 2008 IEEE
DOI 10.1109/ANSS-41.2008.26

ability and the classes of self-stabilizing algorithms it can deal with. As these fault tolerance measures are critical design decision tools, the limitations of the analysis have to be circumvented for large self-stabilizing systems. We therefore propose a simulation framework that can be used to calculate these measures in absence of sufficiently powerful analytic methods.

In this paper, we present a simulation framework called SiSSDA (*Simulator for Self-Stabilizing Distributed Algorithms*) to determine the above mentioned fault tolerance measures. The simulator is written in the programming language Erlang [1, 2]. SiSSDA can simulate self-stabilizing algorithms under different execution semantics and it offers the user functionality to set-up dynamic "fault environments." A fault environment models the temporal and local variations of the transient faults. Under similar fault environments and the same set of distributed algorithms, SiSSDA is able to produce results similar to the results of the analysis procedure as presented for a set of different communication topologies. SiSSDA also overcomes the limitations of the analysis procedure by being able to calculate fault tolerance measures for systems with a larger number of processes under varied fault environments. SiSSDA has been designed in a distributed fashion such that for large systems the load can be shared among multiple computers.

The rest of the paper is organized as follows. In Section 2 we present a short summary of the analytic method along with the system model and compare the properties of existing distributed algorithm simulators with our requirements. The architecture and salient features of SiSSDA are discussed in Section 3. Section 4 contains and discusses the simulation results for two variants of self-stabilizing spanning tree algorithms for a set of network topologies followed by the conclusion in the final section.

2 Preliminaries

In this section, for conciseness of the paper, we briefly review the analytical method of [7] to derive fault tolerance measures of self-stabilizing algorithms and its limitations. We then elaborate on the need of simulation to determine these measures and survey the existing simulators and their usefulness in the application context.

2.1 System Model

A *distributed application* (referred to as *system* interchangeably) constitutes of a finite number of processes, that runs a distributed algorithm, interconnected by a communication infrastructure. A *distributed algorithm* is specified as a set of *sub-algorithms*. Each of the processes executes a sub-algorithm of the distributed algorithm. A system is said to be *self-stabilizing* if and only if, irrespective of the starting state, it reaches the set of safe states within a *finite* number of steps and *remains* in it in the absence of new faults [9].

The system state, that indicates whether the system is in its predefined set of legal states or not, reflects the contents of local variables of the constituting processes and the communication infrastructure.

The underlying *execution semantics* plays an important role while proving the property of self-stabilization of a distributed algorithm. *Serialized semantics* implies that at any time instant only one process takes a step, whereas in *overlapping semantics* a subset of the processes may execute a step at one time instant [3]. An extreme case of overlapping semantics is *maximum parallelism semantics* where *all* the eligible processes in the system execute a step at one time instant [10].

2.2 Dependability Analysis and its Limitations

A method to determine reliability, instantaneous availability and limiting availability of self-stabilizing distributed algorithms has been presented in [7]. *Reliability* for a self-stabilizing system is defined as the probability that the system satisfies its safety predicate at time instant k without ever violating it in time period $[0, k]$. *Instantaneous availability* of a self-stabilizing system at time instant k is defined as the probability that the safety predicate holds provided it was satisfied at time instant 0. *Limiting availability* is a probability that the system satisfies the safety predicate as k tends to ∞. In contrast to the definition of reliability, the definition of availability allows the system to be perturbed by intermittent bursts of failures during system lifetime. The notions of reliability and availability for the self-stabilizing systems are based on the assumption that a self-stabilizing system is in "up-phase" when it satisfies its safety predicate. It is trivial to infer from the definition of self-stabilizing systems that they do not have predefined initial states. An implication of this property is that the set of possible initial states is equal to the complete state space of the system, which is, in the most general case, a *denumerably infinite* set. Hence, the calculation of the dependability measures becomes intractable as all the transitions must be accounted for during the analysis. This problem is circumvented in [7] by partitioning the state space of the algorithm using a method similar to *predicate abstraction* [11]. The state space of the algorithm is partitioned into *configuration classes* where each configuration class is characterized by a predicate over the global state of the algorithm. Such partitioning of the state space implies that for the *silent* self-stabilizing algorithms the set of safe states is represented by a single configuration class. The global predicate can in turn be composed of local predicates defined over each process executing the algorithm. For *locally-checkable* self-stabilizing algorithms [4] with n processes, this method leads to an abstract state space of 2^n configuration classes. Unlike the definition of *execution* used in literature [9], the notion of *observed execution* is used during the analysis. An observed execution allows manifestation of a transient fault as a fault step along with the transitions due to computation by a process, called computation steps, in the system. In order to determine the probabilities of transitions between different configuration classes, each sub-algorithm class of a distributed algorithm is analyzed. The probability that a transition takes a generic process from a state satisfying the local predicate $\mathcal{P}_1$ to a state satisfying the local predicate $\mathcal{P}_2$ is determined for each pair of local predicates. Serialized execution semantics is assumed during this phase of the analysis and the scheduler is modeled via a tuple of probabilities. A Markov chain characterizing the abstracted system is then constructed with help of the probabilities derived as described. The matrix representing the Markov chain is next transformed to account for fault steps caused by the transient faults. The resultant Markov chain is used to determine the desired fault tolerance measures. If required please refer to [7] for a detailed description of the analytic method.

Although the abstraction technique used in the analysis alleviates the state space management problem, it leads to inaccuracy in the measures obtained. The accuracy of the results can be improved by increasing the granularity of the partitioning, however it makes the analysis computationally demanding. In addition to that, the analysis also assumes *pessimistic* error propagation in the system. It implies that, *irrespective* of the fact whether a value read by a process is used in computation or not, a process reading an erroneous value itself is considered erroneous. This, in turn, implies that the values of the fault tolerance measures derived represent lower bounds. Additionally, the analysis method is not directly suited for the self-stabilizing algorithms whose safety condition can not be expressed as state predicate. Unfortunately, the analysis still does not scale well because the number of nodes in the Markov chain increases exponentially with the number of processes in the system.

2.3 Specification of the Simulator

In light of the limitations of the dependability analysis discussed above, the quest for a simulation framework becomes apparent where the values of the fault tolerance measures of a self-stabilizing algorithm under a given execution environment can be evaluated for substantially larger applications and a larger class of of application scenarios. As the algorithm is executed as such, an inaccuracy introduced by the abstraction techniques would not affect the values of the fault tolerance measures. Furthermore, the simulator should be able to determine the fault tolerance measures for the whole class of self-stabilizing distributed algorithms. An implication of this requirement is that the developer should be able to specify customized global predicate-checkers. In order to cope with systems having a large number of processes, the simulator should be able to run simulations in a distributed manner if required. This also implies that subsets of potentially spatially separated processes may be combined to share one common machine for simulation while other subsets are distributed on other machines.

Generally, serialized semantics is used while proving the correctness of the algorithm whereas overlapped semantics is used to study its behavior in an implementation, though the property may not hold under altered semantics [3]. However, some distributed algorithms exhibit the self-stabilization property independent of the underlying semantics. Thus, the simulation framework should be able to simulate the algorithm under any one of the execution semantics.

Due to foreseen requirements of fault model driven model checking, a facility to incorporate environmental sources of defect is also required for future analysis. Hence, the simulator has to be built as a framework to easily adapt future algorithms, system models, fault models, schedulers, execution semantics and environmental influences.

2.4 Related work

We now review existing distributed algorithm simulators.

The *Distributed Algorithms Platform* (DAP) [5] provides an environment for simulation and testing of distributed algorithms. It also provides a graphical user interface for visualizing and controlling the execution of algorithms. DAP implements each simulated process as a separate operating system process and thus offers the possibility of running simulations in a distributed manner. The modular architecture of DAP allows to abstract away details such as the communication infrastructure in the "topology layer." However, it is not possible to add customized "observers" to check whether the simulated algorithm satisfies a particular predicate. In addition, it does not provide functionality to simulate a distributed algorithm under different execution semantics.

SimUTC [21] is a C++SIM-based toolkit for the simulation of fault-tolerant distributed systems. SimUTC is used to simulate round-based clock synchronization algorithms for distributed systems. The framework primarily consist of three modules, namely, `Controller`, `EvalSys` and `Network`. The `Controller` module, as the name implies, is the "lynch pin" of the framework and controls the simulation. `EvalSys` is the GUI which acts as front-end of the `Controller` and performs data analysis. The underlying communication infrastructure is abstracted by the `Network` module. All the modules manifest themselves as separate processes in the C++SIM framework and the simulation is run on a single workstation. SimUTC is able to simulate clock synchronization algorithms, which form a very specific class of distributed algorithms. Also, it is not feasible to simulate distributed algorithms employing a large number of nodes because distributed simulation is not possible.

A framework for the simulation of distributed algorithms built upon *OPNET* [13] and *Xplot* has been presented in [15]. OPNET acts as the simulation engine and Xplot is used to provide graphical output. The three layer modeling hierarchy of OPNET is used to specify the algorithm and the execution environment to be simulated. The "network domain" is used to specify the network topology and the "node domain" describes the architecture of the individual nodes. The distributed algorithm itself is presented to the "process domain" as a finite state machine. But this simulation framework also suffers from the lack of scalability and the functionality to simulate customized execution semantics.

LYDIAN is a framework for simulation and visualization of distributed algorithms primarily used for education purposes [16]. LYDIAN uses an entity called "experiment" to specify the distributed algorithm and the execution environment. An experiment comprises of a "protocol" to be simulated, associated with a "network structure" and a "trace file." The algorithm is specified in a language with C-like constructs. However, as the target group is students, this simulator does not provide functionality to determine fault tolerant measures of the simulated algorithm.

While the simulators described above provide graphical user interface to process the collected data, they lack the capabilities required to derive fault tolerance measures. Also, none of the available distributed algorithm simulators satisfies the requirements outlined. Hence, we set out to develop a simulation framework tailored to our needs. In the following section, we explain the architecture and functionality of the result.

3 The Simulation Framework SiSSDA

We will now explain the modular architecture of SiSSDA followed by the functionalities offered by it. We also elaborate on the format in which information about the algorithm and the fault environment is provided to the simulator.

3.1 Architecture of the Simulator

The basic layout is shown in Figure 1. SiSSDA is designed to run in both, distributed as well as emulated distributed environments. It is implemented in the purely functional language Erlang [2] that supports embedding of foreign programming languages. The target user of the simulation framework is a self-stabilizing systems designer. SiSSDA follows a modular approach as this offers a high degree of flexibility to the user. It primarily consists of four modules: *server*, *fault injector*, *client* and *fault environment*. We next explain the functions of the first three modules while the latter is discussed in Section 3.2 as it is more complex.

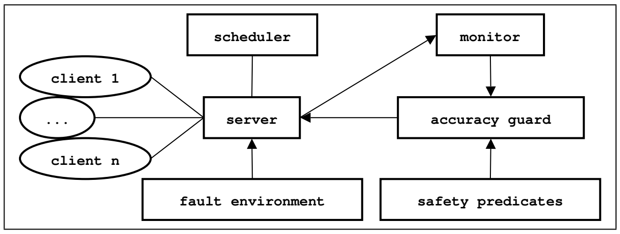

Figure 1. SiSSDA architecture

Server Module The server is the backbone of the simulation framework and controls the entire simulation. It chooses the processes to be activated in each execution step by incorporating an interface to choose between several schedulers. A single process or a subset of processes may be *enabled* by the server depending on the execution semantics specified by the user.

In order to determine whether the safety predicate of a self-stabilizing algorithm is satisfied or not, the local state of every simulated process is collected after each execution step. E.g., if a process is granted one execution step by the scheduler, it might be perturbed by the fault injector. To determine the global system state, each process reports it's local state to the monitoring server module that calculates whether the global safety predicate is satisfied or not.

The server module also provides two possibilities to define a stop criteria: (1) the user can define the exact number of steps to be executed or (2) define *accuracy guards* that stop the simulation automatically as soon as a certain bound of accuracy has been reached. These accuracy guards are further discussed in Section 3.2.

Fault Injector The task of simulating the encompassing implementation scenario is entrusted to the fault injector. The fault injector is, despite some schedulers, the only module that uses a random number generator.

During a simulation, the behavior of the fault injector is specified by the desired fault environment (see Section 3.2). The fault injector waits for the signal from the enabled processes, i.e., from processes that were granted an execution step, and then, depending on the fault environment, decides whether the enabled process should be perturbed or not. If a process is chosen to be corrupted, the fault injector writes an erroneous value to a subset of it's memory register. Note that the values written by the fault injector may be equal to the process' current local state (then acting like a *benign fault*).

Client Module The client module implements the distributed algorithm under investigation. It can be run on a single or on multiple workstations depending upon the number of processes and the structure of the system model to be simulated. The mapping of a client module to a sub-algorithm is done by the server module based on the configuration file during the initialization phase. By default, a client is always in "listening" mode and it becomes active when it receives an enabling signal from the server. An enabled, non-fault-injected client reads the states of its neighbors and computes its own local state based on the values read. Perturbed nodes are directly provided an erroneous value which is used during calculation of the current state. Each node stores the derived own value as well as the values provided by each neighbor. Depending on the definition of the local safety predicates, two functionalities are implemented in SiSSDA: (1) states of neighboring nodes (stored in memory cells) are disregarded and a node satisfies the local safety predicate if the correct value is derived, or (2) not only the current state, but also all neighbors' states (in memory cells) are accounted for calculation of predicate satisfaction. The latter case is more pessimistic as errors might persist even if every node could derive its correct state.

3.2 Features of SiSSDA

SiSSDA can be adapted to provide certain fault tolerance measures for all classes of self-stabilizing algorithms under various fault environments and precision of the obtained measures can be controlled by the user. These classes also cover self-stabilizing systems that are beyond the analysis for feasibility reasons. In the sequel, we describe SiSSDA functionalities and capabilities in more detail.

Fault Environment and Fault Model In order to provide a high granularity in possible errors, faults belong to one of two classes. In the *fault model*, errors occurring during execution due to prescribed behavior (like failed communication or erroneous components) are described. Whether such an error occurs is determined *a priori*, i.e. before execution of a step. Such an error perturbs the execution step and might corrupt the system state.

On the other hand, environmental influences like temperature, lighting or external feedback components might also influence the system without being influenced by the system. Using the data structure, these faults are applied *a postiori*, i.e., they affect the system state after execution of a step, and are accordingly independent of the fault model. These environmental faults are called *fault environment* (in contrast to the fault model). The fault environment, as previously mentioned in Section 3.1, is not only able to corrupt the system state, but for future investigation, regarding for example the scheduling behavior, it can also change the scheduling behavior, the degree of parallelism, the whole system structure, error probabilities and the algorithm applied. The measures observed as well as the accuracy guards, as discussed below, are not prone to the fault environment. Notably, the fault environment is a feature of SiSSDA that was not used during this work as it was neither used in the analytic method.

The simulator can be used to model various concrete fault environment scenarios with the help of the error probability distributions. SiSSDA offers functionality to specify error probabilities for the links and the nodes in the simulated systems. It is also possible to simulate entities with time-variant error probability distributions. For instance, the time-variant error probability distributions for both, links and nodes, can be used to simulate a wireless sensor network. One can also provide a different error probability distribution for each entity to simulate heterogeneous networks.

Even though real systems might require the consideration of such (dynamic) effects, in our cause, to be as close as possible to the analytic approach for comparison reasons, we used a static fault model and set the fault environment to neutral behavior.

Execution Semantics The size of the subset of processes enabled by the server in each execution step is specified by the user. Thus, SiSSDA can simulate the algorithms under a wide variety of execution semantics varying from serialized semantics on one end of the spectrum to maximum parallelism semantics on the other hand. In addition, simulations can also be run under a probabilistic scheduler where the user specifies the probability of a process being enabled in an execution step for each process in the system. This feature can be harnessed to run simulations even under an *unfair* daemon. Furthermore, accumulating effects can be accommodated as schedulers might disregard nodes, e.g. due to some specified hardware wearout.

Accuracy Guard The precision of the fault tolerance measures calculated by SiSSDA can be controlled with help of *accuracy guards*. An accuracy guard primarily specifies the number of digits past the decimal point to which a fault tolerance measure must "stabilize" before it can be reported as the result of a simulation. An accuracy guard is defined by (1) the *minimum number of execution steps*, (2) the *last n calculations* of the desired fault tolerance measure, and (3) the *maximum acceptable deviation* within the set of the last n results. An accuracy guard is "active" when all three conditions are satisfied. Clearly, a stricter accuracy guard leads to longer simulation times while on the other hand a weaker accuracy guard ensures that the simulation ends sooner. However, in the latter case the accuracy of the result obtained may not be too high. Another observation is that same accuracy guards might return results of different accuracy for different scenarios. We conducted some experiments to see the validity of these observations. We simulated the self-stabilizing mutual exclusion algorithm [8] for an eight-process topology (cf. Figure 2) and monitored its limiting availability for the first $20,000$ steps. The fault environment and the fault model featured the same fault probability for all the nodes with no further side effects like parallel execution semantics or change of the scheduler. The result of the simulation is shown in Figure 3. It is easy to see that the variation of availability decreases with the length of the simulation run. We simulated the self-stabilizing mutual exclusion algorithm under two different accuracy guards to show the necessity of a sufficient strictness of the accuracy guards applied. Figure 4 shows the values obtained during these experiments. While for sufficiently strict accuracy guards, the fault tolerance measure observed (namely availability) decreases exponentially with an increase of error probability, the values obtained for weak accuracy guards are not accurate enough. Hence, the graph marked by the squares shows an unsteady progress.

Monitoring Simulation In some cases, it might be important to trace the simulation. SiSSDA provides a *logging* facility such that the execution trace can be analyzed after the end of the simulation. This feature is particularly useful when one is interested in identifying those illegal states which are attained frequently and this knowledge can further be used to re-engineer the system in order to increase

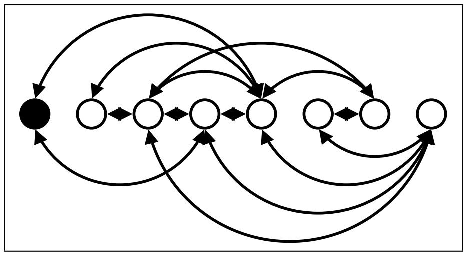

Figure 2. An eight-process topology

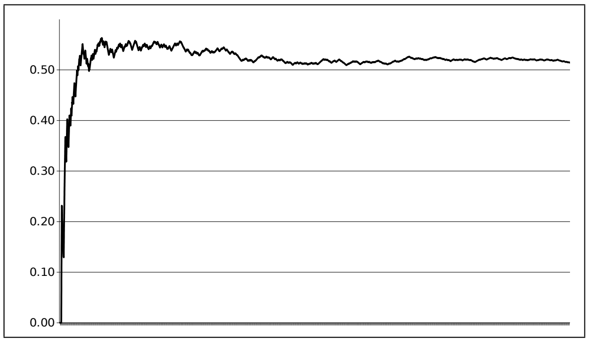

Figure 3. Limiting availability measured in the course of the first 20.000 steps

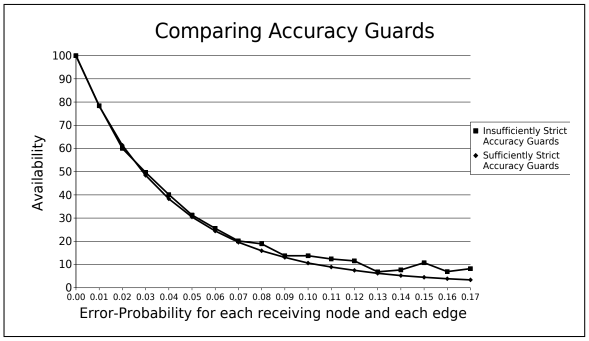

Figure 4. Results determined depends on strictness of accuracy guards employed

some critical dependability measures. In addition, SiSSDA can also be run in *verbose* mode which is interesting from the instructional point of view.

Input Specification A scenario is specified in a configuration file which is read by the server module at the beginning of a simulation. A scenario specifies the distributed algorithm to be simulated along with the desired simulation environment. It is composed of four parameters. The first parameter links the algorithm which is to be simulated. The second parameter specifies the topology according to which the processes communicate. The third parameter specifies the accuracy guard or the number of steps to be executed, respectively. The error probability distribution is given by the fourth parameter. The distributed algorithm is provided through external Erlang modules which are specified in the configuration file.
A detailed description is given in the SiSSDA manual [18] where class diagrams, additional functionalities and ideas for further extension are also presented.

4 Application of the SiSSDA

In this section, we present the results of experiments carried out with SiSSDA. In order to compare the results of the simulations with the analytic approach of [7], we set up exactly the same scenarios for both, simulation and analysis.
For the scenarios, we chose two variants of self-stabilizing spanning tree distributed algorithms along with a set of eleven different topologies. Although the algorithm itself is rather simple, it forms the core of link management protocols and, thus, is critical for smooth network operations [19]. Each topology consists only of one distinct and three common nodes and bidirectional communication links modeled via communication registers. As shown in Figure 5 there are exactly eleven topologies that meet these conditions. To show the advantages of the simulator, we also chose a scenario with an eight-process topology, that is infeasible for the analytical approach. To show the scalability of the simulation framework, we further analyzed the runtime behavior within a distributed environment depending on the number of nodes equally distributed on a different

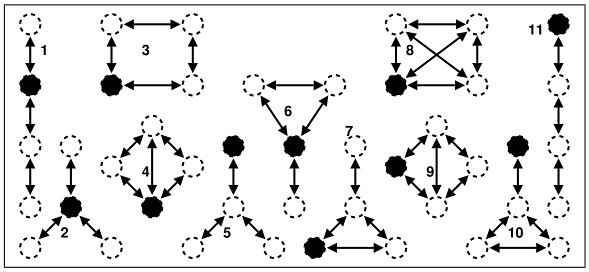

Figure 5. Eleven four-process topologies

number of machines. This comparison is discussed in Section 4.3. The fault-tolerance measure we chose to observe is limiting availability. We describe the algorithms in the next section.

4.1 Case Studies

As explained in the introduction, one of the primary goals for deriving the fault tolerance measures is to compare different self-stabilizing solutions of the same problem. Towards that end, we chose two versions of the solution for the self-stabilizing spanning tree problem. We compared the breadth-first search (BFS) spanning tree algorithm of [9] and depth-first search (DFS) spanning tree algorithm of [6].

```
do
    true  $\rightarrow$  write($path_1 := \bot$)
od
```

Figure 6. DFS sub-algorithm for the root process p_1, from [6].

```
do
    $j \notin \{0,\dots,\delta\}$  $\rightarrow$  $j :=$
        random value in $\{0,\dots,\delta\}$
    $j \in \{0,\dots,\delta\}$  $\rightarrow$
    do
        $j \in \{0,\dots,\delta-1\}$  $\rightarrow$  $j :=$
            $j+1$;  read($read\_path_j := path_{\alpha_i(j)}$)
        $j = \delta$  $\rightarrow$  $j := 0$;  write($path_i :=$
            $\min\{|read\_path_l \circ \alpha_i(l)|_N, 1 \le l \le \delta\}$)
    od
od
```

Figure 7. DFS sub-algorithm for a non-root process $p_i, i > 0$, from [6]

The DFS spanning tree algorithm (see Figures 6 and 7) builds the rooted DFS tree of a graph in self-stabilizing manner. The graph consists of a special node called *root node* and all other nodes are called *non-root nodes*. Every node has a local variable called *path* which encodes the resultant tree. A path is a sequence of node ids starting with "$\bot$." For each node p_i, the variable $path_i$ encodes the path from p_i to the root node. The root node always assigns "$\bot$" to its $path_i$ variable. All other nodes 1) read the path variable of all their neighbors, 2) concatenate their own id to the shortest path variable, and 3) assign the concatenated shortest path variable to their path variable. The system is in a safe state if and if only all the path variables contain the shortest path to the root node.

```
do
    $m \notin \{0,\dots,\delta\}$  $\rightarrow$  $m :=$ random value in
        $\{0,\dots,\delta-1\}$
    $m \in \{0,\dots,\delta-1\}$  $\rightarrow$  $m :=$
        $m+1$;  write($r_{1m} := (0,0)$)
    $m = \delta$  $\rightarrow$  $m := 0$
od
```

Figure 8. BFS sub-algorithm for the root process p_0, from [9]

The BFS spanning tree algorithm (see Figures 8 and 9), likewise, builds the rooted BFS tree of a graph with a root node and multiple non-root nodes. Each process has a distance variable which contains the minimum distance from the root node to that node. In addition, each process owns communication registers which consist of two fields: 1) a distance field and 2) a parent field. The distance field of each communication register contains a copy of its own distance variable. The parent field is set to 1 for the communication register representing the link between a node and its parent. The root repeatedly writes 0 to both fields of its communication registers. Each non-root node 1) chooses the minimum of the distance variables of its neighbors, 2) increments it by 1, and 3) assigns it to its own distance variable. It then 4) copies this value to all its communication registers and 5) the parent field of the communication register representing the link to the neighbor with least distance variable is set to 1.

Both, analysis and simulation, were done with a static fault environment. The fault tolerance measures have been derived under serialized semantics as both algorithms are self-stabilizing under these semantics [9]. We also used uniform error probability with respect to the processes in the system implying that all the nodes were equally vulnerable to faults. This is referred to as *global node error probability* (GNEP) in the following. We next describe the set-up of the simulations for the scenarios described above.

4.2 Running the Simulations

Running the simulator consists of three steps. (1) Change the scenario by editing the file *global_config.hrl* as described in the enclosed *README of [18]*, i.e., set the algorithm to be simulated, the topology, the GNEP and the hostnames as required. To keep compiling times low it is advised to prune such included topologies that will not be simulated. (2) The source code needs to be compiled with *erlc *.erl* from the shell. (3) Start one shell in server mode. After starting another shell as fault injector, simulation commences until it gets interrupted by the user or it reaches the accuracy guards (cf. Section 3.2).

The version of the simulator supplied is designed to run in a multi-processor environment. It will automatically acquire

```
do
  $l \notin \{1,2\} \vee m \notin \{0,\ldots,\delta\} \rightarrow l :=$ random value in $\{1,2\}$;  $m :=$ random value in $\{0,\ldots,\delta-1\}$
  $l \in \{1,2\} \wedge m \in \{0,\ldots,\delta\} \rightarrow$
  do
    $l = 1 \wedge m \in \{0,\ldots,\delta-1\} \rightarrow m := m+1$;  $lr_{mi} := \texttt{read}(r_{\alpha_i(m)i})$
    $l = 1 \wedge m = \delta \rightarrow l := 2$;  $m := 0$;  $\mathit{FirstFound} := \mathit{false}$
    $l = 2 \wedge m \in \{0,\ldots,\delta-1\} \rightarrow$
      $m := m+1$;  $\mathit{dist} = 1 + \min\{lr_{li}.\mathit{dis} \mid 1 \leq l \leq \delta\}$
      if
        $\neg \mathit{FirstFound} \wedge lr_{mi}.\mathit{dis} = \mathit{dist} - 1 \rightarrow \mathit{FirstFound} := \mathit{true}$;  $\texttt{write}(r_{i\alpha_i(m)} := (1, \mathit{dist}))$
        $\mathit{FirstFound} \vee lr_{mi}.\mathit{dis} \neq \mathit{dist} - 1 \rightarrow \texttt{write}(r_{i\alpha_i(m)} := (0, \mathit{dist}))$
      fi

    $l = 2 \wedge m = \delta \rightarrow l := 1$;  $m := 0$
  od
od
```

Figure 9. BFS sub-algorithm for a non-root process $p_i, i > 0$, from [9]

spare resources. Client processes are automatically created by the server node.

4.3 Results

The GNEP was varied from 0.00 to 0.40 via incremental steps of 0.05. The GNEP was not increased beyond 0.40 because the limiting availability converges to 0 and does not change much after a GNEP of 0.40. Figures 10 and 11 show the variation of availability with respect to the GNEP for the analytical approach. The variation of limiting availability determined by the simulations is shown in Figures 12 and 13. The deviation of limiting availability for a single scenario across ten experiments was sufficiently small as shown in Figures 14, 15 and 16. Each simulation was run for one million execution steps which is sufficient as discussed at the end of this section.

The decrease of limiting availability follows an exponential pattern for both, the analysis and the simulation. Also, the DFS spanning tree algorithm outperforms the BFS spanning tree algorithm in both algorithms. This can be attributed to the fact that the BFS spanning tree algorithm uses a far larger number of variables than the DFS algorithm, namely

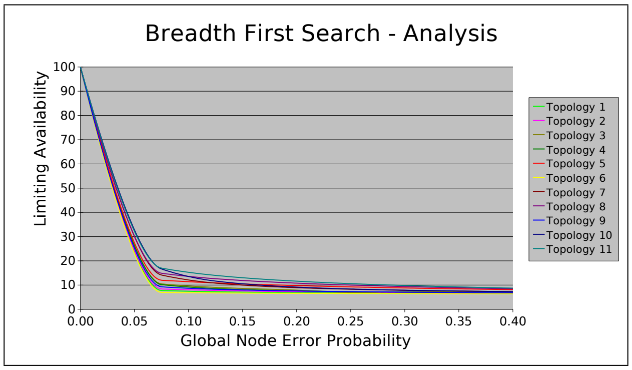

Figure 10. BFS analysis results

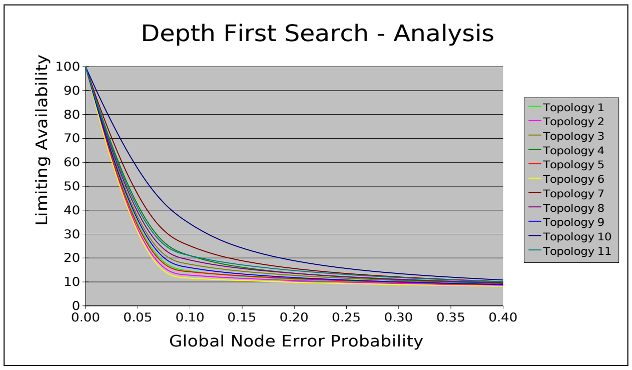

Figure 11. DFS analysis results

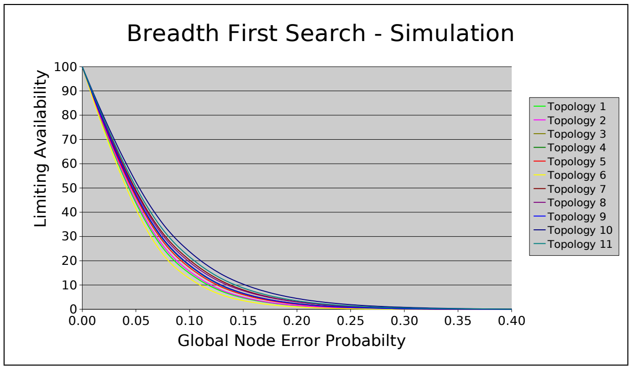

Figure 12. BFS simulation results

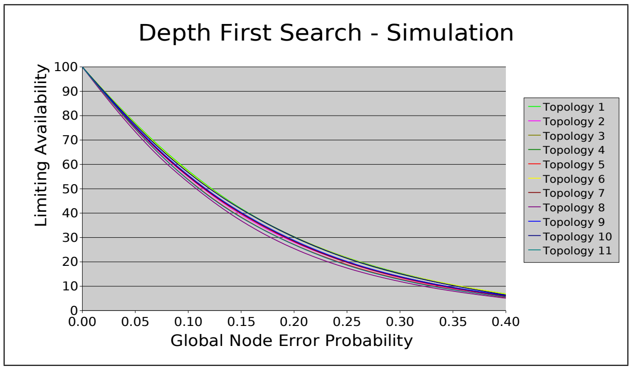

Figure 13. DFS simulation results

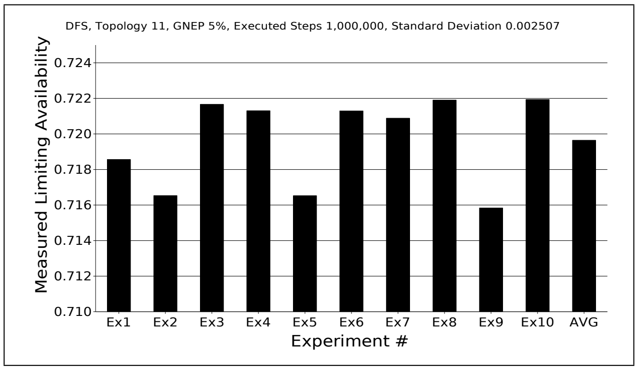

Figure 14. Reliability of results with GNEP 0.05

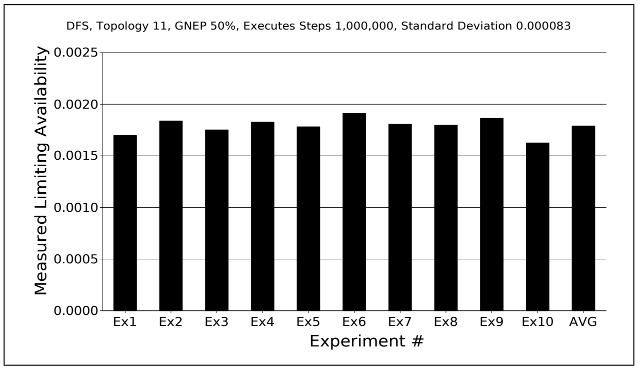

Figure 15. Reliability of results with GNEP 0.5

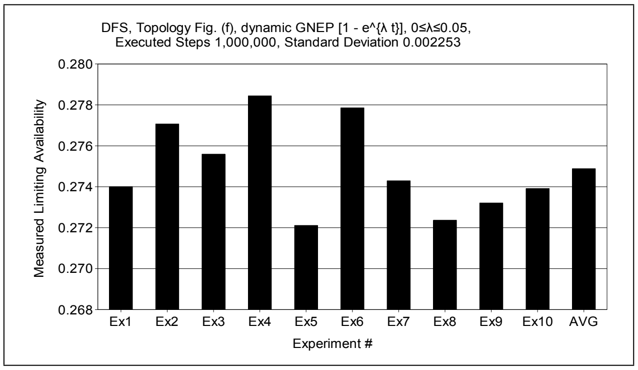

Figure 16. Reliability of results with dynamic GNEP

the memory registers, and thus all these variables must be in a correct state such that the safety predicate is satisfied. The variation of limiting availability among different topologies for the same value of GNEP can be attributed to the *connectivity* of the respective topology. The more connected the topology is, the higher is the impact of *error propagation*. The limiting availability derived by the analysis method is bounded by the number of configuration classes used to construct the respective Markov chain. Thus, for a four-process system, it is bounded by 0.125 as shown in the plots. The limiting availability derived analytically approaches the value derived during simulation if the number of configuration classes is increased. However, this makes the analytic method extremely computationally demanding. We also simulated both self-stabilizing spanning tree algorithms on an eight processes system topology (see Figure 2) to gauge the scalability of SiSSDA. Figure 17 shows the result of the simulations for this system. In order to further test the scalability of the simulator, we ran simulated algorithms on topologies with large numbers of nodes. We simulated the self-stabilizing BFS algorithm on random graphs

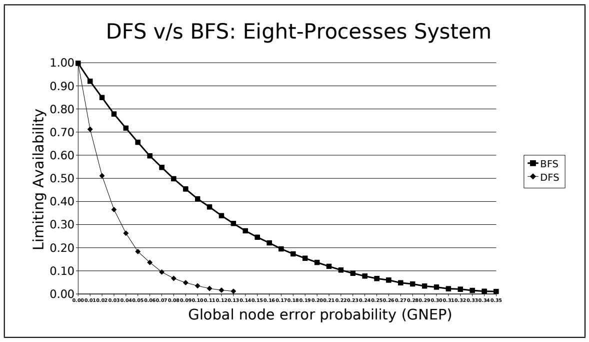

Figure 17. Simulation of an eight-process topology

with 64, 128, 256, 512 and 1024 nodes. It is advised to run each client, server and fault injector application on a different machine if the number of simulated processes is above a certain threshold. For instance, during our experiments a machine with 1GB of main memory could not simulate a topology with more than 256 nodes, on the other hand a topology with 1024 nodes was handled by a machine with 4GB memory.

All the simulations were executed on both, a single system and multiple systems, and time required to meet the accuracy guards for each scenario was compared. The time required for the simulation is proportional to the size of the topology. If the simulator is run over local network, overall computation time increases considerably because computation time is small in comparison with message passing time. Nevertheless, it is advantageous to run the simulator in distributed fashion in order to simulate very large topologies. Thus, the number of nodes that can be simulated can be arbitrarily large modulo the number of systems available for simulation. Also, running time decreases linearly for computationally demanding algorithms.

As shown in Figures 14 and 15, results can be reproduced with a sufficient accuracy regardless of the GNEP applied. Since accuracy margins are relative to the GNEP, variance in results is negligible.

5 Conclusion

We presented a simulation framework to determine fault tolerance measures for self-stabilizing algorithms to overcome the limitations of the currently available analytic method. SiSSDA is able to calculate the fault tolerance measures for self-stabilizing systems under various implementation scenarios. SiSSDA has a modular design which provides flexibility with respect to the specification of algorithms. We explained constituting modules of SiSSDA and described their functionalities. The results of simulations for two self-stabilizing algorithms under similar assumptions were compared to corresponding results of the analysis and it has been shown that fault tolerance measures derived by simulation were comparable to that of the analysis procedure. We further showed the capabilities of SiSSDA which go beyond the capabilities of the analysis procedure. SiSSDA is able to derive fault tolerance measures for self-stabilizing systems using a considerably larger number of processes. It is able to simulate self-stabilizing algorithms which the analytic method is currently not able to handle.

References

[1] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007.

[2] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[3] P. C. Attie, N. Francez, and O. Grumberg. Fairness and Hyperfairness in Multi-Party Interactions. *Distributed Computing*, 6(4):245 – 254, 1993.

[4] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-Stabilization By Local Checking and Correction (Extended Abstract). In *FOCS*, pages 268 – 277. IEEE, October 1991.

[5] I. Chatzigiannakis, A. Kinalis, A. Poulakidas, G. Prasinos, and C. Zaroliagis. DAP: A Generic Platform for the Simulation of Distributed Algorithms. In *ANSS*, pages 167 – 177, April, 2004.

[6] Z. Collin and S. Dolev. Self-Stabilizing Depth-First Search. *Information Proc. Letters*, 49(6):297–301, 1994.

[7] A. Dhama, O. Theel, and T. Warns. Reliability and Availability Analysis of Self-Stabilizing Systems. In *SSS*, volume LNCS 4280, pages 244 – 261. Springer, 2006.

[8] E. W. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Communications of ACM*, 17(11):643 – 644, 1974.

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

[10] M. G. Gouda and T. Herman. Stabilizing Unison. *Inf. Process. Lett.*, 35(4):171–175, 1990.

[11] S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *CAV*, number 1254 in LNCS, pages 72–83, 1997.

[12] T. Herman. Models of Self-Stabilization and Sensor Networks. In *IWDC'03*, volume LNCS 2918, pages 205 –214. Springer-Verlag, 2003.

[13] OPNET Technologies Inc. OPNET Modeler, 1999.

[14] P. Jalote. *Fault Tolerance in Distributed Systems*. Prentice-Hall, 1994.

[15] S. Khanvilkar and S. M. Shatz. Tool Integration for Flexible Simulation of Distributed Algorithms. *Software - Practice and Experience*, 31(14):1363 – 1380, November 2001.

[16] B. Koldehofe, M. Papatriantafilou, and P. Tsigas. LYDIAN: An extensible educational animation environment for distributed algorithms. *ACM Journal on Educational Resources in Computing*, 6(2):1 – 21, June 2006.

[17] N. Mitton, E. Fleury, I. Guérin Lassous, and S. Tixeuil. Self-Stabilization in Self-Organized Multihop Wireless Networks. In *ICDCS Workshops*, pages 909–915. IEEE Computer Society, 2005.

[18] N. Müllner. *SiSSDA Manual* `http://www.informatik.uni-oldenburg.de/~phoenix/docs/Manual.pdf`. Germany, April 2007.

[19] R. J. Perlman. An algorithm for distributed computation of a spanningtree in an extended LAN. In *SIGCOMM*, pages 44–53, 1985.

[20] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley and Sons Ltd., 2nd edition, 2002.

[21] B. Weiss, G. Gridling, U. Schmid, and K. Schossmaier. The SimUTC Fault-Tolerant Distributed Systems Simulation Toolkit. In *MASCOTS*, pages 68 – 75. IEEE, 1999.