Evaluating Practical Tolerance Properties of Stabilizing Programs through Simulation: the Case of Propagation of Information with Feedback

Jordan Adamek[1], Mikhail Nesterenko[1], and Sébastien Tixeuil[2⋆]

[1] Kent State University
[2] UPMC Sorbonne Universités & IUF

Abstract. We simulate a stabilizing propagation of information with feedback (PIF) program to evaluate its response to perturbations. Under several classic execution models, we vary the extent of the fault as well as the system scale. We study the program's speed of stabilization and overhead incurred by the fault. Our simulation provides insight into practical program behavior that is sometimes lacking in theoretical correctness proofs. This indicates that such simulation is a useful research tool in studies of fault tolerance.

1 Introduction

System stabilization [2, 12, 8, 1] of various flavors is an attractive approach to optimistic failure recovery. A system is stabilizing if, regardless of the initial state, it eventually arrives at a legitimate state. This allows the program to recover from any fault, regardless of its nature, after the influence of the fault stops. This property allows researchers in this area to effectively ignore the nature of the fault. Classic papers on stabilization usually contain the algorithm description, proof of its correctness, and performance bound estimates.

Extensive performance evaluations are relatively rare in the field [4–7, 13]. If stabilization simulation is carried out, the researchers tend to focus on time of algorithm recovery from randomly chosen initial global states. We believe that such studies give an incomplete picture of robust algorithm behavior. A random initial state is expected to represent a systemic fault. However, uniformly random states tend to present a rather mild challenge to the algorithm as individual process states end up being evenly distributed across the state space. Moreover, such initial states may not adequately represent the states that occur after faults influence legitimate states of the algorithm. In effect, random initial states tend to hide the complexity of faulty behavior and resultant algorithm recovery. Another simplification is the focus on recovery time, a staple of stabilization research. We believe that other characteristics, such as the expense of

⋆ This work was supported in part by ANR project SHAMAN.

recovery in terms of fault-induced extra steps, is also an important characteristic of a stabilizing algorithm.

In this paper, we describe the performance evaluation of a stabilizing propagation of information with feedback [11] algorithm. Variants of such algorithm have been extensively studied in stabilization literature. Taking this algorithm as an example, we studied its recovery from faults of increasing scale under classic execution semantics. We measured the algorithm's stabilization time as well as its stabilization overhead. The result is a detailed chart of the algorithm's behavior across fault scales and system sizes.

2 Algorithm Description

The algorithm implements propagation of information with feedback (PIF) for rooted trees. Each process has access to the read-only variables *parent* and Ch that respectively contain the identifier of the parent, and the set of identifiers of the children for this process. Each process can be in one of the three states: idle, requesting and replying. This state is encoded in the variable *st* as **i**, **rq**, and **rp** respectively. The root can only be requesting or idle while a leaf can be either idle or replying. Each process can read the state of its neighbors and update its own in a single atomic step. The actions of the algorithm are shown in Figure 1. Let us consider any chain of processes from the root to a leaf and possible legitimate states in this chain. In case the request has not reached the leaf, the chain starts with a possibly empty sequence of requesting processes, followed by at least one idle process and concluding by a possibly empty sequence of replying processes that belong to the previous request. Once the request has reached the leaf, the chain starts with a non-empty sequence of requesting processes followed by a non-empty chain of replying processes. This algorithm is proven self- and ideally-stabilizing [8, 9]. That is, regardless of the initial state, the algorithm achieves one of the above legitimate states in a specified sequence and remains in a legitimate state afterwards. A variant of this algorithm is proven snap-stabilizing [1]. That is, the algorithm transitions to a legitimate state within a single wave cycle.

$$
\begin{array}{lllll}
\mathit{request}: & & st.root = \mathbf{i} & \wedge\ (\forall q \in Ch.root : st.q = \mathbf{i}) & \longrightarrow st.root := \mathbf{rq} \\
\mathit{clear}: & & st.root = \mathbf{rq} & \wedge\ (\forall q \in Ch.root : st.q = \mathbf{rp}) & \longrightarrow st.root := \mathbf{i} \\
\mathit{forward}: & st.parent = \mathbf{rq} & \wedge\ st.p = \mathbf{i} & \wedge\ (\forall q \in Ch.p : st.q = \mathbf{i}) & \longrightarrow st.p := \mathbf{rq} \\
\mathit{back}: & st.parent = \mathbf{rq} & \wedge\ st.p = \mathbf{rq} & \wedge\ (\forall q \in Ch.p : st.q = \mathbf{rp}) & \longrightarrow st.p := \mathbf{rp} \\
\mathit{stop}: & st.parent = \mathbf{i} & \wedge\ st.p \neq \mathbf{i} & & \longrightarrow st.p := \mathbf{i} \\
\mathit{reflect}: & st.parent = \mathbf{rq} & \wedge\ st.leaf = \mathbf{i} & & \longrightarrow st.leaf := \mathbf{rp} \\
\mathit{reset}: & st.parent = \mathbf{i} & \wedge\ st.leaf = \mathbf{rp} & & \longrightarrow st.leaf := \mathbf{i}
\end{array}
$$

Fig. 1. PIF algorithm actions. Actions *request* and *clear* belong to the root process; actions *forward*, *back*, and *stop* – to an intermediate processes; actions *reflect* and *reset* – to a leaf.

3 Experiment Description and Results

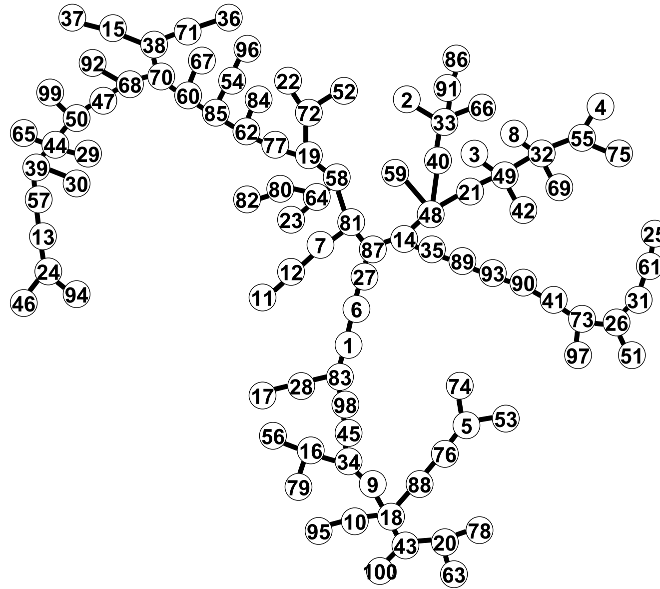

Fig. 2. Example random tree.

Tree generation The PIF algorithm evaluation requires rooted tree selection. Unbiased tree selection is a non-trivial task. We use Prüfer sequences [10]. A labeled sequence of size $n-2$ uniquely defines one of all possible trees of n nodes. Hence, selecting a uniformly random labeled sequence of length $n-2$ gives each tree an equal probability of being chosen. An example tree is shown in Figure 2. For the generated tree, we select the root uniformly at random.

Initial state selection. For the initial state of the algorithm computation, we select a legitimate state and then perturb it by a fault. The initial state selection for PIF also requires care: in the execution of the algorithm, certain states may appear more often than others. For example, in a tree of large size, in most of the states, the root is requesting. Indeed, once the root receives the feedback from one wave, it immediately starts the next one. Hence, the global states where the root process is idle are rare. Due to uneven occurrence of global state in a computation, the states are not evenly exposed to faults. In other words, faults are more likely to occur in states of the algorithm that happen more often. To generate an initial legitimate state, we randomly select the number of execution steps between zero and the number that is ten times the system size. We then start the algorithm from the legitimate initial state where all processes are idle

and run the algorithm for the selected number of steps. Thus obtained state is used as the initial state for our experiment.

Execution semantics. To produce the computations of our algorithm we implement three classic algorithm execution semantics (also known as schedulers or daemons [3]): interleaving (centralized), powerset (distributed) and (maximally) synchronous. To produce the next state of the computation, for each execution semantics we evaluate each process to determine which actions are enabled. The execution semantics differ by the selection of enabled actions for execution. For *interleaving semantics*, we randomly select one of the enabled actions to execute. For the other two semantics the selection procedure is more complicated. The algorithm is proven correct in interleaving semantics only. Thus, the actions of two neighbor processes cannot be executed in a single step without compromising the correctness of the algorithm. The enabled action selection is as follows. We randomly choose one of the enabled actions. After that, we eliminate the neighbor process actions from consideration and repeat the selection. For *powerset semantics* the number of selected enabled actions is chosen at random between 1 and the total number of enabled actions. For *synchronous semantics* we continue the selection until no more actions remain.

Selection of fault model. To measure the robustness of our algorithm, we introduce faults in the initial state of the system. In a faulty process, the state is randomly selected from the range of possible states. Note that the fault may perturb the process back into the legitimate state. In other words, a fault may have no observable effect. This method sounds counterintuitive. However, it precludes the correct state of the process from influencing the faulty state. If every process is faulty, the resultant state is completely random.

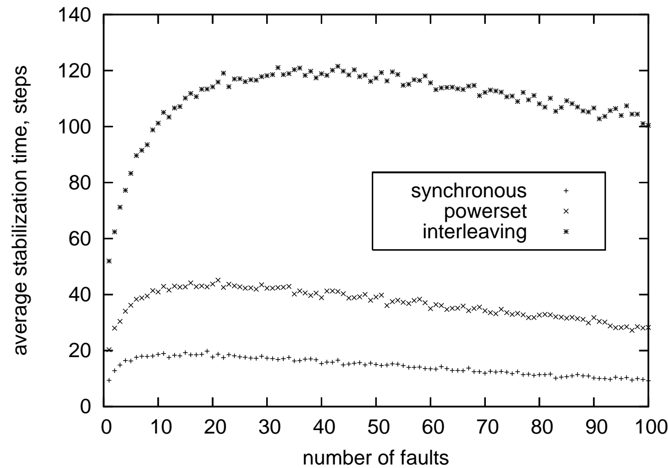

Fig. 3. Stabilization time.

Stabilization time and overhead. We focus on two stabilization metrics: stabilization time and processing overhead. The *stabilization time* is the number of execution steps it takes the algorithm to achieve the legitimate state. The *overhead* is the number of extra action executions that the fault induces the algorithm to perform. To describe out definition of overhead, let us revisit a chain of processes from the root to a leaf. Even in an illegitimate state there is an non-empty sequence of processes that starts with the root and conforms to the definition of PIF legitimacy. Let us consider the longest such sequence. An action is counted as overhead if it is executed outside this sequence and not counted as overhead otherwise. In other words, the overhead actions are those that the algorithm executes before it achieves the legitimate state excluding the actions involved in the root request propagation. In the literature on snap-stabilization [1], for the interleaving execution semantics this metric is called wait time.

Experiments. We ran two sets of experiments. We used the system of 100 processes and varied the number of faults from one to 100. For each number of faults, we run $1,000$ experiments. In each experiment we selected a random tree and a random initial state for it. The generated trees had the average height of 21.6 ± 4.9 and the average number of children was 37.5 ± 3.1. As we varied the faults, we calculated the stabilization time and overhead for the three execution semantics. The results are shown in Figures 3 and 4. In the second set of experiments, we fixed the particular fault rate and varied the scale of the system. The results are shown in Figure 5.

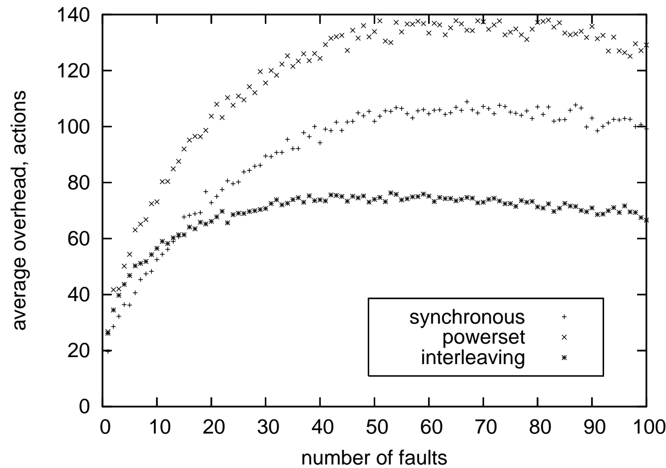

Fig. 4. Processing overhead.

Analysis. The simulation results present a detailed picture of PIF fault recovery behavior. As the number of faults increases, the stabilization time and overhead rises and then gradually subsides. This seems counterintuitive. However, further investigation indicated the reason for such algorithm behavior. If a state is legitimate, a single fault may initiate a spurious wave that runs in the opposite direction to the legitimate wave. This happens, for example, if the reply propagates towards the root and the fault changes the state of one of the replying processes to idle. This may also happen if the wave propagates towards the leaves and the fault switches the state of one of the requesting processes to replying. Stabilization from such faults takes time proportional to the system size. Further faults tend to break up these long spurious waves and decrease stabilization time.

The overhead and stabilization time of parallel and power-set execution semantics is lower than that of the interleaving semantics. The execution semantics that allow greater concurrency lead to faster stabilization and thus lower overhead. As the scale set of experiments indicates, as the size of the system grows, the stabilization time grows linearly. Similarly to the first experiment set, the growth for the completely random initial state (100% fault rate) is lower than for the smaller number of faults. For 1% rate, the stabilization time is about twice the system size, while for 100% fault rate, it is about one and a half.

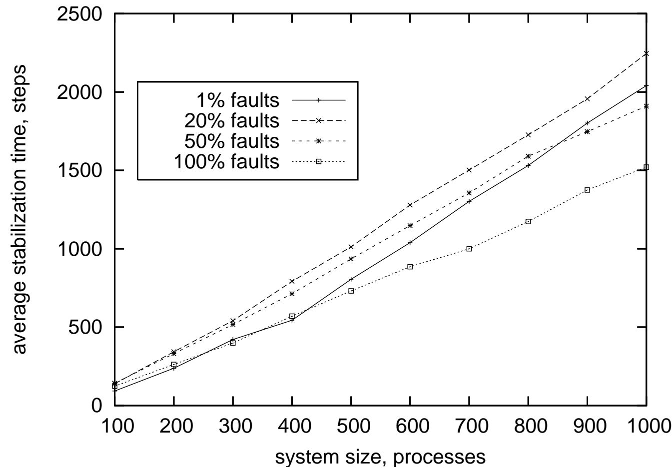

Fig. 5. Stabilization time dependence on system size. Interleaving semantics.

4 Conclusion

The performance of PIF is by no means exhaustive. For example, depending on its location, a fault affects stabilization of the algorithm differently. A fault closer to the root may be a lot more detrimental since it has greater potential to spread state corruption throughout the tree. Similarly, estimating the reliability of our simulation with confidence intervals would be useful. More extensive investigation of PIF performance could be the subject of further research.

However, performance evaluation of the stabilizing PIF algorithm in this paper makes a case for more extensive simulation studies of stabilizing algorithms. We believe this greater engagement with applied algorithm study would benefit stabilizing algorithm design itself. Once the researchers observe the algorithm behavior in the case of practical faults, they will construct stabilizing algorithms that are specifically designed to counteract such realistic conditions. This will lead to greater applicability of stabilizing research overall.

References

1. A. Bui, A.K. Datta, F.Petit, and V. Villain. Snap-stabilization and PIF in tree networks. *Distributed Computing*, 20(1):3–19, 2007.
2. Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.
3. Swan Dubois and Sébastien Tixeuil. A taxonomy of daemons in self-stabilization. Technical Report 1110.0334, ArXiv eprint, October 2011.
4. M Flatebo and AK Datta. Simulation of self-stabilizing algorithms in distributed systems. In *Proceedings of the 25th Annual Simulation Symposium*, pages 32–41, 1992.
5. Colette Johnen and Fouzi Mekhaldi. Self-stabilization versus robust self-stabilization for clustering in ad-hoc network. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, volume 6852 of *Lecture Notes in Computer Science*, pages 117–129. Springer Berlin / Heidelberg, 2011.
6. Nathalie Mitton, Bruno Séricola, Sébastien Tixeuil, Eric Fleury, and Isabelle Guérin-Lassous. Self-stabilization in self-organized multihop wireless networks. *Ad Hoc and Sensor Wireless Networks*, 11(1-2):1–34, January 2011.
7. 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.
8. Mikhail Nesterenko and Sébastien Tixeuil. Ideal stabilization. *Journal of Utility and Grid Computing (JUGC)*, 2012. to appear.
9. Mikhail Nesterenko and Sébastien Tixeuil. Proof of stabilization of ideal propagation of information with feedback algorithm. Technical Report TR-KSU-CS-2012-1, Computer Science Department, Kent State University, April 2012.
10. A. Prüfer. Neuer beweis eines Sätzes über permutationen. *Archiv für Mathematik Physik*, 27:122–142, 1918.
11. G. Tel. *Introduction to distributed algorithms*. Cambridge University Press, 1994.
12. Sebastien Tixeuil. *Algorithms and Theory of Computation Handbook, Second Edition*, chapter Self-stabilizing Algorithms, pages 26.1–26.45. CRC Press, Taylor & Francis Group, November 2009.

13. Sally K. Wahba, Jason O. Hallstrom, Pradip K. Srimani, and Nigamanth Sridhar. SFS ^{3}: a simulation framework for self-stabilizing systems. In Robert M. McGraw, Eric S. Imsand, and Michael J. Chinni, editors, *Proceedings of the 2010 Spring Simulation Multiconference, SpringSim 2010, Orlando, Florida, USA, April 11-15, 2010*, pages 172–181. SCS/ACM, 2010.