Message-efficient Self-stabilizing Transformer using Snap-stabilizing Quiescence Detection

Anaïs Durand and Shay Kutten

Technion - Israel Institute of Technology, Haifa, Israel
danais@technion.ac.il, kutten@ie.technion.ac.il

Abstract. By presenting a message-efficient snap-stabilizing quiescence detection algorithm, we also facilitate a transformer that converts non self-stabilizing algorithms into self-stabilizing ones. We propose a message-efficient snap-stabilizing ongoing quiescence detection algorithm. (Notice that by definition it is also self-stabilizing and can detect termination.) This algorithm works for diffusing computations. We are not aware of any other self-stabilizing or snap-stabilizing ongoing quiescence or termination detection algorithm.

Keywords: Fault-tolerance · Snap-stabilization · Quiescence · Termination · Diffusing computations

1 Introduction

Self-stabilization [11] is a property of distributed systems that withstand transient faults. After transient faults set it into an arbitrary state, a self-stabilizing system recovers in finite time a correct behavior. Multiple transformers that transform non self-stabilizing algorithms $\mathcal{A}$ into self-stabilizing ones [2, 4–6] works roughly as follows. First, $\mathcal{A}$ is executed. When $\mathcal{A}$ terminates, a local checking algorithm is executed (called "local detection" algorithm [2] or local *verifier* of a *Proof Labeling Scheme* [19]). This verifier detects an illegal state if and only if a fault occurred. A self-stabilizing *reset* algorithm, *e.g.*, [3], is then executed to bring all the nodes to an initial state that is legal for $\mathcal{A}$. The cycle is then started again, *i.e.*, $\mathcal{A}$ is executed, termination detected, *etc.* Note that a proof labeling scheme has to be designed especially for $\mathcal{A}$, and some change to $\mathcal{A}$ may be needed in order to generate the specific proof labeling scheme.

The above transformers assume a *synchronous* network in order to know that $\mathcal{A}$ terminated and the verifier could be activated to verify the output (otherwise, the verifier would signal a fault since the output is not yet computed). We do not want this assumption. Alternatively [18], such transformers use a self-stabilizing synchronizer [3,7]. This is a very message intensive function. It uses $\Omega(m)$ messages per round (where m is the number of edges). For example, if $\mathcal{A}$'s time complexity is $\Omega(n)$, its self-stabilizing version (using such a transformer), would need $\Omega(nm)$ messages till stabilization. An earlier transformer uses even more messages [17]. (It assumed a self stabilizing leader election, which was then provided by [2]). The snap-stabilizing quiescence detection algorithm presented here is a much more message-efficient termination detection method in place of the self-stabilizing synchronizer, at least for diffusing computations [12] (*e.g.*, DFS, BFS, token circulation).

Still, one needs yet another component for the transformer. Indeed, since $\mathcal{A}$ is not self-stabilizing, if a fault occurs, $\mathcal{A}$ may never terminates. The missing component is one that enforces termination. Here we use a very simple enforcer: assume that a node sends at most some x messages executing $\mathcal{A}$ when there is no fault. To implement the enforcer, each node just refuses to send more than x messages. It turns out that even under these constraints (diffusing computations and the simple enforcer) the resulting transformer sends less messages than the traditional ones for various algorithms.

Quiescence detection. A distributed system reaches *quiescence* [9, 21] when no messages are in the communication links and a local indicator of stability holds at every process. Termination and deadlock are two examples of quiescence properties. Detecting quiescence is fundamental. When a deadlock is detected, some measures can be taken such as initiating a reset. Detecting the termination of a task allows the system to use its computed result or issue another operation. In particular, a distributed application is often composed of several modules where one must wait for the termination of a module before starting the next one. It is considered easier to design a task that eventually terminates and combine it with a termination detection protocol, see [15].

The quiescence detection problem and its sub-problems have been extensively studied in distributed computing since the seminal works of Dijkstra and Scholten [12] and Francez [14] on termination detection. One can distinguish two main kinds of quiescence detection algorithms. *Ongoing detection* algorithms must monitor the execution since its beginning and eventually detects quiescence when it is reached, *e.g.*, [12]. *Immediate detection* algorithms answers whether the system has reached quiescence by now or not, *e.g.*, [14]. Ongoing quiescence detection is needed for the transformer, and for most other applications. Ongoing detection can be designed using an immediate detection algorithm by repeatedly executing the detection algorithm until it actually detects quiescence, however it might be highly inefficient.

Cournier *et al.* [10] explain how to design a snap-stabilizing[1] immediate termination detection algorithm using their *Propagation of Information and Feedback (PIF)* algorithm in the locally shared memory model. This does not seem applicable for the message efficient transformer - not only this is not an ongoing detection, the memory requirement is large since the whole state of the system must be locally computed and stored (this can also increase the message complexity in the $\mathcal{CONGEST}$ model).

Contributions. We propose the first self-stabilizing and snap-stabilizing ongoing quiescence detection algorithm $\mathcal{Q}$ for *diffusing computations*.[2] Using $\mathcal{Q}$, we also implement a message-efficient self-stabilizing transformer.

$\mathcal{Q}$ requires $O(\Delta \log n)$ bits per process, where Δ is the maximum degree. The additional cost of $\mathcal{Q}$ is $O(t_{ab} + m_{\mathcal{A}} + n)$ rounds, where t_{ab} is the number of rounds needed to empty all the initial messages out of the channels and reach stabilization of

[1] *Snap-stabilization* [8] is a variant of self-stabilization that ensures immediate recovery after transient faults. Notice that a snap-stabilizing algorithm is also self-stabilizing.

[2] In a *diffusing computation*, a unique process, the *initiator*, can spontaneously send a message to one or more of its neighbors and only once. [12]. After receiving their first message, the other processes can freely send messages to their neighbors.

the alternating bit protocol of Afek and Brown [1]. The message complexity of $\mathcal{A}$, the monitored algorithm, is denoted $m_{\mathcal{A}}$.

2 Quiescence Detection Algorithm $\mathcal{Q}$

We assume the $\mathcal{CONGEST}$ model [20] with FIFO channels of message capacity one (see [1, 4] to enforce this).

A *(global) quiescent* property is defined by a *local quiescent-indicator* $quiet(p)$ at each process p such that: (a) while $quiet(p)$ holds, p does not send messages and, as long as p does not receives a message, $quiet(p)$ continues to hold; (b) the channels are empty and $quiet(p)$ holds at every process p if and only if quiescence is reached.

In the context of snap-stabilization (see [8]), a quiescence detection algorithm can start from an arbitrary configuration that leads processes to signal quiescence even if quiescence is not actually reached. In particular, some message can initially be in some channel (p, q) while neither p or q are aware of it until q receives it. Thus, processes have two output signals: $SignalQ()$ and $SignalE()$. A process calls $SignalQ()$ when it detects (global) quiescence. $SignalE()$ is called when an error is detected, *i.e.*, the execution did not start from a *clean configuration*. For example, in a clean configuration of our algorithm $\mathcal{Q}$ it is required, among other things, that channels are empty and an execution of $\mathcal{A}$ starting from this configuration is actually a diffusing computation.

Definition 1. *$\mathcal{Q}$ is a* snap-stabilizing ongoing quiescence detection *algorithm if, for every execution Γ where $\mathcal{Q}$ monitors algorithm $\mathcal{A}$ since the beginning of its execution:*

- Eventual Detection: *If the execution of $\mathcal{A}$ reaches quiescence, a process eventually calls $SignalQ()$ or $SignalE()$.*
- Soundness: *If $SignalQ()$ is called, either the execution of $\mathcal{A}$ actually reached quiescence or the initial configuration of $\mathcal{Q}$ was not clean.*
- Relevance: *If the execution of $\mathcal{A}$ satisfies $\mathcal{E}$ and the initial configuration of $\mathcal{Q}$ is clean, no process ever calls $SignalE()$.*

The relevance property prevents a trivial and useless detection algorithm where a process calls $SignalE()$ in every execution. Notice that there is no assumption on $\mathcal{A}$, *i.e.*, we do not require $\mathcal{A}$ to be self-stabilizing or even to compute a correct result.

Overview of the Algorithm. $\mathcal{A}$ and $\mathcal{Q}$ are composed using a fair composition [13]. To avoid confusion, we call *packets* the messages of $\mathcal{A}$. The idea of $\mathcal{Q}$ adapts the algorithm of Dijkstra and Scholten [12] to the snap-stabilizing context using local checking [2]. To monitor $\mathcal{A}$ and detect quiescence, $\mathcal{Q}$ builds the tree of the execution. The initiator of the diffusing computation is the root. When a process that is not in the tree receives a packet m, it joins the tree by choosing the sender of m as parent. When a process p has no children and $quiet(p)$ holds, p leaves the tree. $SignalQ()$ is called when the initiator has no children and its local quietness-indicator holds.

To ensure that quiescence is not signaled when some messages are traveling, $\mathcal{Q}$ uses acknowledgments to wait until messages are received before taking any action of leaving the tree. In [12], counters are used to keep track of how many messages have not been acknowledged yet. In a stabilizing context, maintaining counters is not easy. Thus,

$\mathcal{Q}$ sends and receives packets of $\mathcal{A}$ using a self-stabilizing *alternating bit protocol* [1]. Simple proof labeling schemes [19] are used in various parts of the algorithm to make sure it performs correctly. (Those schemes are somewhat generalized in the sense that they are used to verify properties of the algorithm while the algorithm is still running.) See the full version of the paper.

References

1. Afek, Y., Brown, G.M.: Self-stabilization over unreliable communication media. Distributed Computing **7**(1), 27–34 (1993)
2. Afek, Y., Kutten, S., Yung, M.: The local detection paradigm and its application to self-stabilization. Theor. Comput. Sci. **186**(1-2), 199–229 (1997)
3. Awerbuch, B., Kutten, S., Mansour, Y., Patt-Shamir, B., Varghese, G.: Time optimal self-stabilizing synchronization. In: STOC'93. pp. 652–661 (1993)
4. Awerbuch, B., Patt-Shamir, B., Varghese, G.: Self-stabilization by local checking and correction (extended abstract). In: FOCS'91. pp. 268–277 (1991)
5. Awerbuch, B., Patt-Shamir, B., Varghese, G., Dolev, S.: Self-stabilization by local checking and global reset. In: WDAG'94. pp. 326–339 (1994)
6. Awerbuch, B., Varghese, G.: Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In: FOCS'91. pp. 258–267 (1991)
7. Boulinier, C., Petit, F., Villain, V.: When graph theory helps self-stabilization. In: PODC 2004. pp. 150–159 (2004)
8. Bui, A., Datta, A.K., Petit, F., Villain, V.: State-optimal snap-stabilizing PIF in tree networks. In: WSS'99. pp. 78–85 (1999)
9. Chandy, K.M., Misra, J.: An example of stepwise refinement of distributed programs: Quiescence detection. ACM TOPLAS **8**(3), 326–343 (1986)
10. Cournier, A., Datta, A.K., Devismes, S., Petit, F., Villain, V.: The expressive power of snap-stabilization. Theor. Comput. Sci. **626**, 40–66 (2016)
11. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM **17**(11), 643–644 (1974)
12. Dijkstra, E.W., Scholten, C.S.: Termination detection for diffusing computations. Information Processing Letters **11**(1), 1–4 (1980)
13. Dolev, S.: Self-stabilization. MIT press (2000)
14. Francez, N.: Distributed termination. ACM TOPLAS **2**(1), 42–55 (1980)
15. Francez, N., Rodeh, M., Sintzoff, M.: Distributed termination with interval assertions. In: Formalization of Programming Concepts. pp. 280–291 (1981)
16. Hendler, D., Kutten, S.: Bounded-wait combining: constructing robust and high-throughput shared objects. Distributed Computing **21**(6), 405–431 (2009)
17. Katz, S., Perry, K.J.: Self-stabilizing extensions for message-passing systems. Distributed Computing **7**(1), 17–26 (1993)
18. Korman, A., Kutten, S., Masuzawa, T.: Fast and compact self-stabilizing verification, computation, and fault detection of an MST. In: PODC'11. pp. 311–320 (2011)
19. Korman, A., Kutten, S., Peleg, D.: Proof labeling schemes. Distributed Computing **22**(4), 215–233 (2010)
20. Peleg, D.: Distributed Computing: A Locality-sensitive Approach. Society for Industrial and Applied Mathematics (2000)
21. Shavit, N., Francez, N.: A new approach to detection of locally indicative stability. In: ICALP'86. pp. 344–358 (1986)