Reducing the Number of Messages in Self-stabilizing Protocols

Anaïs Durand
Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
anais.durand@lip6.fr

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

March 25, 2020

Abstract

Self-stabilizing algorithms recover from sever faults, such as inconsistent initialization. Traditionally, when designing a self-stabilizing message-passing algorithm, the main goal was to reduce the time until stabilization. The message cost was neglected. In this work, we strive to reduce the number of messages sent on the average per time period. As a tool, we present a stabilizing module that can message-efficiently determine when a task (from a wide family of tasks) is terminated. False positive detection is possible, but only when faults occurred. This module is then used in the transformation of non self-stabilizing algorithms into self-stabilizing ones.

Keywords: Fault-tolerance · Self-stabilization · Message complexity · Quiescence detection · Termination detection.

1 Introduction

In 1974, Dijkstra [11] introduced the *self-stabilization* as a property of distributed that withstand sever faults. If a self-stabilizing system is led by faults into any incorrect global state, it eventually recovers a correct behavior. For example, the token circulation algorithms proposed in [11] can recover from an arbitrary initial configuration where several processes hold a token instead of only one. After recovery, exactly one token remains. Self-stabilizing protocols for various problems have been devised: leader election, synchronization, *etc.* However, when designing a self-stabilizing algorithm, the message complexity is traditionally neglected and the designers only aim at reducing the stabilization time, *i.e.*, the time before recovering a correct behavior. This happened probably because a self-stabilizing message-passing algorithm cannot stop. It needs to continuously send messages in order to check whether faults occurred and recovery is needed.

In particular, multiple transformers that convert a non self-stabilizing algorithm $\mathcal{A}$ into a self-stabilizing one have been designed [1, 3, 4, 5]. Most of those transformers work roughly as follows. First, $\mathcal{A}$ is executed. Then, once the execution of $\mathcal{A}$ is terminated, a local checking algorithm is executed (called "local detection" algorithm [1] or the local *verifier* of a *Proof Labeling Scheme* [18]). This checking detects when the state is illegal (because a fault occurred). For example, if $\mathcal{A}$ is an algorithm to construct a routing tree of shortest paths (SPT), the verifier checks that the states of every node (but the root) include a parent pointer, and that the collection of parent pointers forms a tree of shortest paths. If a fault is actually detected,

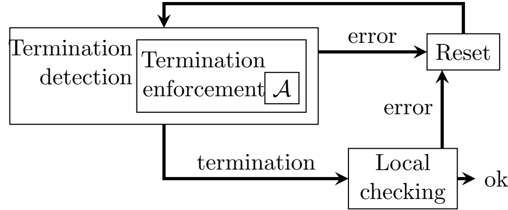

Figure 1: Schematic overview of the proposed transformer.

a self-stabilizing *reset* algorithm, *e.g.*, [2], is executed in order to bring all the nodes to a legal initial state of $\mathcal{A}$. Then, the process starts again, *i.e.*, $\mathcal{A}$ is executed, termination detected, *etc.*[1]

Being able to detect termination is necessary to know when the verifier should be activated. Otherwise, if the verification is done before the output is computed, a fault would be signaled. For example, before algorithm $\mathcal{A}$ of the above example terminates, the SPT is not yet computed so the verifier may interpret that as a fault. The above transformers assume a *synchronous* network to detect that enough time has passed so $\mathcal{A}$ must have terminated. However, we do not want such an assumption. Alternatively [17], such transformers can use a self-stabilizing synchronizer [2, 6]. Unfortunately, a self-stabilizing synchronizer is very costly in number of exchanged messages. 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 (and would continue using $\Omega(m)$ messages forever). An earlier transformer uses even more messages [16]. (It assumed a self stabilizing leader election, which was then provided by [1]).

In this work, we present a method for reducing the number of messages sent on the average per time period (compared to using synchronizers), at least for a wide class of tasks called *diffusing computations* [12] (*e.g.*, DFS, broadcast and echo, two-phase commit, token circulation). In a diffusing computation, a unique process, the *initiator*, can spontaneously send a message to one or more of its neighbors and only once. After receiving their first message, the other processes can freely send messages to their neighbors. Indeed, we propose a snap-stabilizing[2] quiescence detection algorithm tailored to detect when the execution of $\mathcal{A}$ is terminated by proposing a termination detection method more message-efficient than a self-stabilizing synchronizer. Note that both detection methods may have a one sided error. That is, *if* faults occurred, the detector may detect termination even though $\mathcal{A}$ has not terminated; such a false detection is still useful for a transformer, since triggering *reset* to rerun $\mathcal{A}$ would be the right thing to do in the case faults occurred.

Another component is needed for the transformer. If the execution of $\mathcal{A}$ starts in an arbitrary configuration because of faults, it may never terminate since $\mathcal{A}$ is not self-stabilizing. Thus, the transformer needs a mechanism to enforce termination. We can use a very simple enforcer as follows. Assume that an upper bound x on the number of messages that a node sends in $\mathcal{A}$ when there is no fault is known. For example, in broadcast and echo, the number of messages each node sends is bounded by twice the number of its neighbors. To implement the enforcer, each node just refuses to send more than x messages. Figure 1 proposes a schematic overview of the transformer.

Quiescence Detection. *Quiescence* [8] is a global property of distributed systems. A distributed system is quiet when the communication channels are empty and a local indicator of

[1] A proof labeling scheme has to be designed especially for $\mathcal{A}$, and some changes to $\mathcal{A}$ may be needed in order to generate the specific proof labeling scheme.

[2] A *snap-stabilizing* [7] algorithm is a self-stabilizing algorithm that recovers immediately after faults occurred.

stability holds at every process. Termination is an example of quiescence property. Detecting quiescence is a known fundamental problem in distributed computing. For example, in addition to its usefulness in the self-stabilizing transformer, detecting the termination of a task allows the system to know the computed result is ready for output. Moreover, termination detection simplifies the design of a complex task. The task is broken into modules, such that some module m_2 must wait until some other module m_1 terminates. It is easier to design a terminating m_1, and then couple it with a termination detection protocol [14].

Since the seminal works of Dijkstra and Scholten [12] and Francez [13] on termination detection in distributed systems, the quiescence detection and its sub-problems have been extensively studied. For a survey, see [20]. Two main kinds of quiescence detection algorithms can be distinguished. *Ongoing detection* algorithms monitors the execution since its beginning and eventually detects quiescence when it is reached, *e.g.*, [12]. A different approach is the *immediate detection* algorithms that answer whether the system has reached quiescence by now or not, *e.g.*, [13]. 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.

A self-stabilizing *Propagation of Information and Feedback (PIF)* algorithm [10, 19, 21] can be used to design an immediate termination detection algorithm, see [9]. Varghese [21] proposes a self-stabilizing PIF algorithm in the message-passing model. Snap-stabilizing PIF algorithms are proposed in [10, 19]. Such an application would have high communication and memory complexity even without the need to convert this farther to an ongoing detection.

Contributions. First, we propose a new measure for message efficiency for *asynchronous* networks, where we count the number of messages in executions that are "reasonable" synchronous, *i.e.*, *k-synchronous executions*. Then, we propose a self-stabilizing and snap-stabilizing ongoing quiescence detection algorithm $\mathcal{Q}$ for *diffusing computations*. Using $\mathcal{Q}$, one can implement a message-efficient self-stabilizing transformer. When $\mathcal{Q}$ monitors an algorithm $\mathcal{A}$, it detects quiescence or signals an error in $O(t_{\mathcal{A}} + n)$ rounds, where $t_{\mathcal{A}}$ is the round complexity of $\mathcal{A}$ and n is the number of processes. Its memory complexity is $O(\Delta \log n)$ bits per process, where Δ is the maximum degree. If the execution is k-synchronous, the message complexity of the quiescence detection algorithm is $O(k(m + n(t_{\mathcal{A}} + n) + M_{\mathcal{A}}))$, where m is the number communication links and $M_{\mathcal{A}}$ is the message-complexity of $\mathcal{A}$.

Roadmap. In the next section, we detail the considered computational model and the specification of the quiescence detection problem. Section 3 presents our snap-stabilizing quiescence detection algorithm $\mathcal{Q}$ and an analysis of its correctness and complexities is given in Section 4.

2 Preliminaries

Consider connected *distributed systems* of n processes operating in the asynchronous message-passing model. The topology of the system is represented by a graph $\mathcal{G} = (V, E)$ where V is the set of processes and E is the set of communication links. Each process can send messages to and receive messages from a subset of other process called *neighbors*. $\mathcal{N}_p$ denotes the set of neighbors of process p, *i.e.*, $(p, q) \in E \Leftrightarrow q \in \mathcal{N}_p$. Communications are bidirectional, *i.e.*, $p \in \mathcal{N}_q \Leftrightarrow q \in \mathcal{N}_p$. We assume reliable (no message is lost) and FIFO (messages are received in the order they are sent) channels of bounded capacity c. Messages are received in finite time. The size of a message is restricted to $\Theta(\log n)$ bits.

Variables and Executions. Every process has a finite number of variables. Let us denote $p.x$ the variable x of process p. Assume a unique process is distinguished as the initiator of the diffusing computation, *i.e.*, every process p has a constant $p.\mathsf{init}$ that evaluates to `true` at a unique process. The *state* of a process is the vector of the values of its variables. The *state* of a channel is the list of messages it contains. A *configuration* is a vector of states, one for every process or channel in the network. Denote by $\gamma(p).x$ the value of variable $p.x$ in configuration γ.

Let $\mapsto$ be a binary relation over configurations such that $\gamma \mapsto \gamma'$ is a *step*, *i.e.*, γ' can be obtained from γ by the *activation* of one or more processes, *i.e.*, some messages are received and/or sent, some internal computation is done. It is required that during a step, a process receives and sends at most one message over each of its connecting channel. An *execution* is a sequence of configurations $\Gamma = \gamma_0, \gamma_1, \ldots, \gamma_i, \ldots$, such that $\forall i \geq 0$, $\gamma_i \mapsto \gamma_{i+1}$. Configuration γ_0 is the *initial configuration* of Γ. Infinitely often during an execution, a process triggers a timeout and processes it to do some internal computation and/or to send some messages.

A *round* is a unit of complexity measure and is defined as follows. It is the shortest execution such that every message in transit (*i.e.*, inside the channels) at the beginning of the round is received (and processed) by its reciepient and every process triggers (and processes) a timeout.

Quiescence Detection. A *(global) quiescent* property is characterized by a *local quiescence-indicator* $Quiet(p)$ at each process p such that:

- *Quiescence:* If $Quiet(p)$ holds, p does not send messages and, as long as p does not receives a message, $Quiet(p)$ continues to hold.
- *Local Indicativity:* The channels are empty and $Quiet(p)$ holds at every process p if and only if quiescence is reached.

For example, for the termination property, $Quiet(p)$ holds when p is disabled.

A distributed algorithm is *snap-stabilizing* [7] *w.r.t.* some specification S if any execution starting from an arbitrary configuration satisfies S. In this context, we define the set of *regular* initial configurations of the quiescence detection algorithm, *i.e.*, initial configurations where the detection algorithm is well initialized. Notice that, since the initial configuration is arbitrary, it can be not regular. (The definition of regular initial configurations for our algorithm is given in Definition 7.)

The goal of a quiescence detection algorithm $\mathcal{Q}$ is to detect when the execution of another algorithm $\mathcal{A}$ that $\mathcal{Q}$ monitors, reaches quiescence.

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

- Eventual Detection: *If the execution of $\mathcal{A}$ reaches quiescence, some process eventually calls $SigQuiet()$ or $SigError()$.*
- Soundness: *If $SigQuiet()$ is called, either the execution of $\mathcal{A}$ reached quiescence or was not a diffusing computation, or the initial configuration of $\mathcal{Q}$ was not regular.*
- Relevance: *If the execution of $\mathcal{A}$ is a diffusing computation and the initial configuration of $\mathcal{Q}$ is regular, no process ever calls $SigError()$.*

$SigQuiet()$ and $SigError()$ are two output signals. When such a signal is emitted, it triggers an external response from the system, *e.g.*, a reset [1, 2, 3]. Notice that there is no hypothesis on $\mathcal{A}$, *i.e.*, we do not require $\mathcal{A}$ to be self-stabilizing or even to compute a correct result.

3 Quiescence Detection Algorithm $\mathcal{Q}$

In this section, we propose a self-stabilizing ongoing quiescence detection algorithm $\mathcal{Q}$ for diffusing computations written in the message-passing model. The code of $\mathcal{Q}$ is presented in Algorithm 1.

Algorithm 1: Algorithm $\mathcal{Q}$ for Process p

```
1:  upon PIF_rcv(q, ⟨pckt, dist⟩) do
2:     if ¬Error(p) then
3:        p.status := Act ;  Deliver(q, pckt) ;
4:        if ¬p.init ∧ p.par = ⊥ then  p.par := q ;   p.dist := dist + 1 ;
5:     if p.par = q then  PIF_send_fbck(q, ⟨true⟩);
6:     else PIF_send_fbck(q, ⟨false⟩);
7:  upon PIF_fbck(q, ⟨isChild⟩) do
8:     if ¬Error(p) then
9:        p.pckt[q] := ⊥ ;
10:       if isChild then                                          // q is a child of p
11:          if p.status = Act ∧ (p.init ∨ p.par ≠ ⊥) then  p.child[q] := true ;
12:          else  p.status := Err ;   SigError() ;
13:       else  p.child[q] := false;
14: upon rcv(q, ⟨Par⟩) do                              // q thinks it is the parent of p
15:    if ¬Error(p) ∧ p.par ≠ q then  send(q, ⟨NoChild⟩) ;
16: upon rcv(q, ⟨Child, dist⟩) do                                  // q is a child of p
17:    if ¬Error(p) then
18:       if p.status = Act ∧ (p.init ∨ p.par ≠ ⊥) ∧ dist = p.dist + 1 then
19:          p.child[q] := true;
20:       else  p.status := Err ;   SigError() ;
21: upon rcv(q, ⟨NoChild⟩) do                                   // q is not a child of p
22:    if ¬Error(p) then p.child[q] := false;
23: upon timeout do
24:    if Error(p) then p.status := Err ;   SigError() ;
25:    else if Passive(p) then
26:       p.status := Pass ;
27:       if p.par ≠ ⊥ then  send(p.par, ⟨NoChild⟩) ;   p.par := ⊥ ;
28:       if p.init then  SigQuiet();
29:    else
30:       foreach q ∈ N_p : p.pckt[q] ≠ ⊥ do
31:          p.status := Act ;
32:          PIF_send(q, ⟨p.pckt[q], p.dist⟩) ;
33:       foreach q ∈ N_p : p.child[q] = true do  send(q, ⟨Par⟩);
34:       if p.par ≠ ⊥ then  send(p.par, ⟨Child, p.dist⟩);
```

Overview. The idea of $\mathcal{Q}$ adapts the algorithm of Dijkstra and Scholten [12] to the stabilizing context using local checking [1, 3]. To monitor an algorithm $\mathcal{A}$ and detect its quiescence, $\mathcal{Q}$ handles the sending and reception of messages of $\mathcal{A}$, that we will call *packets* in order to avoid confusion. $\mathcal{Q}$ builds the *tree of the execution* defined as follows. The initiator of the diffusing computation is the root of the tree. When a process p receives a packet *pckt*, it joins the tree by choosing the sender of *pckt* as its parent by updating variable p.par. Each process p has also a Boolean variable p.child$[q]$ for each of neighbor q, stating if q is a child of p. When a process p has no children and predicate $Passive(p)$ holds, p leaves the tree by notifying its parent and removing its parent pointer, if p is not the initiator. Otherwise, it signals quiescence.

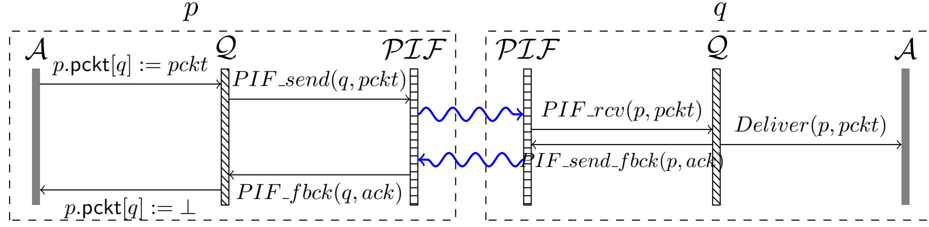

Figure 2: Schematic view of how the packets of $\mathcal{A}$ from process p to process q are handled. The wavy arrows illustrate the communications between p and q through protocol $\mathcal{PIF}$.

Algorithm 2: Macro of Modification of Algorithm $\mathcal{A}$

```
/* Replace every:                                   */
1: send(q, pckt);
/* by:                                              */
2: wait until p.pckt[q] = ⊥;
3: p.pckt[q] := pckt;
```

Handling the messages of $\mathcal{A}$. In order to allow $\mathcal{Q}$ to manage the packets of the monitored algorithm $\mathcal{A}$, we assume that the functions of $\mathcal{A}$ to send and receive packets are slightly altered as shown in Algorithm 2. Every process p has a variable $p.\mathsf{pckt}[q]$ for each neighbor q used to communicate between $\mathcal{A}$ and $\mathcal{Q}$. Indeed, $p.\mathsf{pckt}[q]$ contains the packet of $\mathcal{A}$ that p wants to send to q, or $\perp$ if no packet has to be sent. When p needs to send some packet *pckt* to q in $\mathcal{A}$, p must wait until the previous packet is processed, *i.e.*, until $\mathcal{Q}$ sets $p.\mathsf{pckt}[q]$ to $\perp$. On the other hand, when a process p receives a packet *pckt* from q in $\mathcal{Q}$, this packet is delivered to $\mathcal{A}$ at p using $Deliver(q, pckt)$, *i.e.*, it triggers a $rcv(q, pckt)$ in $\mathcal{A}$.

In order to ensure that the packets of $\mathcal{A}$ are delivered and quiescence is not signaled when some messages are in transit, $\mathcal{Q}$ uses a snap-stabilizing *Propagation of Information with Feedback (PIF)* algorithm [10], denoted here $\mathcal{PIF}$. A PIF allows a process to send a messages to other processes (*propagation*) and to receive in return an acknowledgment from those other processes (*feedback*). Let us use $\mathcal{PIF}$ as follows. To send packets to a neighbor q, a process p uses a dedicated instance of $\mathcal{PIF}$ involving only p and q. Notice that one instance of $\mathcal{PIF}$ between only two processes requires a constant number of bits per process and so $O(\Delta)$ bits per process to send and receive packets from all neighbors (where Δ is the maximum degree). Primitives of the PIF algorithm are prefixed with *PIF_*. When a message *msg* is sent from p to q through the $\mathcal{PIF}$ protocol (*i.e.*, using $PIF_send(q, msg)$), it triggers a $PIF_rcv(p, msg)$ at process q. Then q sends a feedback to p using $PIF_send_fbck(p, ack)$ that triggers a $PIF_fbck(q, ack)$. Notice that the messages of $\mathcal{Q}$ used to do the detection are not transmitted through $\mathcal{PIF}$ since no feedback on those message is required.

Those three protocols – $\mathcal{A}$, $\mathcal{Q}$, and $\mathcal{PIF}$ – are composed using a fair composition. Figure 2 illustrates how the packets of $\mathcal{A}$ are handled and the interactions between the three protocols.

Quiescence and Error Detection. To check whether the execution tree is correctly built and to update the knowledge of a process about its children, every process p regularly send control messages along the tree: $\langle \text{PAR} \rangle$ to its children (Line 33) and $\langle \text{CHILD}, p.\mathsf{dist} \rangle$ to its parent (Line 34), where $p.\mathsf{dist}$ is the distance from p to the root. In particular, if a process q receives a message $\langle \text{CHILD}, dist \rangle$ from one of its children p and $dist \neq q.\mathsf{dist} + 1$, the distances in the tree are not well computed (*e.g.*, the tree contains a cycle) so q signals an error. If a process q receives a message $\langle \text{PAR} \rangle$ from a neighbor p that is not its parent, it sends back a message $\langle \text{NOCHILD} \rangle$ and p can update its variable $p.\mathsf{child}[q]$.

In addition, process p locally checks the correctness of the tree, *i.e.*, if predicate $LCorrect(p)$ holds. For example, p can check that it has no children if it is not attached to the tree.

(Notice that the formal definition of $LCorrect(p)$ is given in Definition 3.) If $Error(p) \equiv (\neg LCorrect(p) \vee p.\mathsf{status} = \text{Err})$ holds, p signals an error (Lines 24-24). A process p which calls $SigError()$ also sets $p.\mathsf{status}$ to Err.

A process p leaves the tree when it becomes *passive*, *i.e.*, the following predicate $Passive(p)$ holds.

Definition 2. *Predicate $Passive(p)$ holds if $Quiet(p)$ holds,*[3] *p has no packets to send and no received packets to process (*i.e.*, there is no incoming event PIF_rcv and $\forall q \in \mathcal{N}_p$, $p.\mathsf{pckt}[q] = \bot$), and p has no children (that it knows of),* i.e.*, $\forall q \in \mathcal{N}_p, \neg p.\mathsf{child}[q]$.*

4 Analysis

In this section, we show that $\mathcal{Q}$ is a snap-stabilizing ongoing quiescence detection algorithm and we analyze its complexities. Due to the lack of space, some straightforward proofs are omitted. Notice that since the three algorithms $\mathcal{A}$, $\mathcal{Q}$, and $\mathcal{PIF}$ are fairly composed, a process executes a round of one of these algorithms every 3 rounds.

4.1 Properties of the PIF protocol

First, let us state some useful property of the snap-stabilizing PIF protocol $\mathcal{PIF}$ from [10].

Proposition 1. *If a process p initiates a PIF to send a message m to its neighbor q through the protocol $\mathcal{PIF}$ (*i.e.*, using $PIF_send(q, m)$), it is received by q ($PIF_rcv(p, m)$) in at most 8 rounds of $\mathcal{PIF}$. Then, p receives the feedback from q in at most 1 additional round of $\mathcal{PIF}$ ($PIF_fbck(q, fbck)$). Moreover p cannot receive a feedback from q before q received m.*

Notice that it does not prevent situations where the arbitrary initial configuration generates faulty communications of the $\mathcal{PIF}$ protocol leading to some process p receiving feedback without initiating any PIF. However, once p actually initiated some PIF with a neighbor q, *i.e.*, the call to function $PIF_send(q, m)$ is terminated and $\mathcal{PIF}$ will manage the communications between p and q to ensure the transmission of m, p cannot receive any feedback that is sent by q before q receives m. Let *tainted messages* be messages of $\mathcal{PIF}$ present in the arbitrary initial configuration or generated afterwards and that are not related to a PIF *actually* initiated by some process.

Proposition 2. *If there are no tainted messages in the channels (*i.e.*, in transit or in the incoming mailboxes of processes but not yet processed by their recipient), a process p cannot receive any feedback from a neighbor q if p does not initiate any PIF to send a packet to q.*

Since one PIF lasts at most 9 rounds of $\mathcal{PIF}$ (by Proposition 1), we can deduce the following corollary.

Corollary 1. *After $9 \times 3 = 27$ rounds, there is no more tainted messages.*

4.2 Execution Trees

Now, let us show that par-variables actually define trees. If the structure of the tree is incorrect, process p locally detects the errors using predicate $LCorrect(p)$ defined as follows.

Definition 3. *Predicate $LCorrect(p)$ holds at some process p if all of the four following conditions are satisfied:*

[3] Predicate $Quiet(p)$ is defined in Section 2.

$C_1(p)$. $(p.\mathsf{status} \neq \text{Err} \wedge p.\mathsf{init}) \Rightarrow (p.\mathsf{par} = \bot \wedge p.\mathsf{dist} = 0)$
$C_2(p)$. $(p.\mathsf{status} \neq \text{Err} \wedge \neg p.\mathsf{init} \wedge p.\mathsf{par} = \bot) \Rightarrow (\forall q \in \mathcal{N}_p : \neg p.\mathsf{child}[q])$
$C_3(p)$. $(p.\mathsf{status} \neq \text{Err} \wedge \neg p.\mathsf{init} \wedge p.\mathsf{par} = \bot) \Rightarrow (\forall q \in \mathcal{N}_p : p.\mathsf{pckt}[q] = \bot)$
$C_4(p)$. $p.\mathsf{status} = \text{Pass} \Rightarrow p.\mathsf{par} = \bot$

Lemma 1. *Let $p \in V$. If the execution of $\mathcal{A}$ is a diffusing computation, we have the following results.*

1. *Let $\gamma \mapsto \gamma'$. If $LCorrect(p)$ holds in γ then $LCorrect(p)$ holds in γ'.*
2. *In at most 3 rounds, $LCorrect(p)$ holds.*

Now, let us prove that par-variables actually define trees. We define three conditions on the relationship between p and its parent $q \in \mathcal{N}_p$.

Definition 4. *Let $p \in V$ and $q \in \mathcal{N}_p$.*
$C_5(p, q)$. $(p.\mathsf{status} \neq \text{Err} \wedge q.\mathsf{status} \neq \text{Err} \wedge p.\mathsf{par} = q) \Rightarrow (q.\mathsf{child}[p] \vee q.\mathsf{pckt}[p] \neq \bot)$
$C_6(p, q)$. $(p.\mathsf{status} \neq \text{Err} \wedge q.\mathsf{status} \neq \text{Err} \wedge p.\mathsf{par} = q) \Rightarrow (q.\mathsf{init} \vee q.\mathsf{par} \neq \bot)$
$C_7(p, q)$. $(p.\mathsf{status} \neq \text{Err} \wedge q.\mathsf{status} \neq \text{Err} \wedge p.\mathsf{par} = q) \Rightarrow p.\mathsf{dist} = q.\mathsf{dist} + 1$

Lemma 2. *Let $p \in V$ and $q \in \mathcal{N}_p$.*

1. *Let $\gamma \mapsto \gamma'$ s.t. no tainted messages are in the channels. If $C_5(p, q)$, $C_6(p, q)$, and $C_7(p, q)$ hold in γ then $C_5(p, q)$, $C_6(p, q)$, and $C_7(p, q)$ hold in γ'.*
2. *In at most 33 rounds, $C_5(p, q)$, $C_6(p, q)$, and $C_7(p, q)$ hold.*

Definition 5. *The* execution graph $\mathcal{G}_{ex} = (V_{ex}, E_{ex})$ *is the subgraph of non-error processes induced by* par*-variables,* i.e., $V_{ex} = \{p \in V : p.\mathsf{status} \neq \text{Err}\}$ *and* $E_{ex} = \{(p, q) \in E : p.\mathsf{par} = q \wedge p, q \in V_{ex}\}$.

From Lemmas 1 and 2, one can deduce the following corollary.

Corollary 2. *In at most 33 rounds, $\mathcal{G}_{ex}$ is a forest.*

4.3 Detection of Quiescence

In this subsection, we show that $\mathcal{Q}$ fulfills the three properties of quiescence detection: eventual detection (Theorem 1), soundness (Theorem 2), and relevance (Theorem 3). Let γ_0 be the initial configuration and let $\gamma_0(\mathcal{A})$ be the projection of γ_0 on the variables and messages of $\mathcal{A}$.

Theorem 1 (Eventual Detection)**.** *If any execution of $\mathcal{A}$ starting from $\gamma_0(\mathcal{A})$ reaches quiescence, then a process calls $SigError()$ or $SigQuiet()$ in $O(t_{\mathcal{A}} + n)$ rounds, where n is the number of processes and $t_{\mathcal{A}}$ is the maximum number of rounds for $\mathcal{A}$ to reach quiescence from $\gamma_0(\mathcal{A})$.*

Proof. First, notice that if the execution is not a diffusing computation, *i.e.*, if some process p that is not the initiator of $\mathcal{A}$ spontaneously requires the sending of some packet of $\mathcal{A}$ in $\mathcal{Q}$ (*i.e.*, $p.\mathsf{pckt}[q]$ becomes different than $\bot$ with $q \in \mathcal{N}_p$), p signals an Err (see Line 24). If $\mathcal{Q}$ signals an error, we are done. Now, assume that $SigError()$ is never called.

Since every packet of $\mathcal{A}$ is eventually delivered through $\mathcal{PIF}$ and thanks to the fair composition, $\mathcal{Q}$ does not block the execution of $\mathcal{A}$. Let γ be the first configuration after which $\mathcal{G}_{ex}$ is a forest (see Corollary 2) and $\mathcal{A}$ reached quiescence, *i.e.*, at every process p, $Quiet(p)$ holds and p does not require the sending of a packet of $\mathcal{A}$ ($\forall q \in \mathcal{N}_p$, $p.\mathsf{pckt}[q] = \bot$).

When some process p requires the sending of a packet in $\mathcal{A}$ to some neighbor q, *i.e.*, $p.\mathsf{pckt}[q] \neq \bot$, p initiates a PIF (Line 32) in at most 3 rounds. The PIF lasts 27 rounds (Proposition 1) before p sets $p.\mathsf{pckt}[q]$ to $\bot$ and allows the sending of the next packet of $\mathcal{A}$. Thus, in at most $30t_{\mathcal{A}}$ rounds, every packet of $\mathcal{A}$ has been delivered. So γ is reached in at most

$33t_{\mathcal{A}} + 27$ rounds. Notice that no process has status ERR or it would have signaled an error in the meantime (Lines 24).

Now, let us show that the height of the trees decreases in at most 12 rounds and that eventually every process has status PASS. First, since $\forall p \in V$, $\forall q \in \mathcal{N}_p$, $p.\mathsf{pckt}[q] = \bot$, no PIF is initiated after γ. Hence, no process can get a PIF_rcv anymore. Let p be a leaf in $\mathcal{G}_{ex}$. By definition, $\forall q \in \mathcal{N}_p$, $\gamma(q).\mathsf{par} \neq p$. Since p will never send a packet to q, $q.\mathsf{par}$ cannot become equal to p anymore. Now, let us show that $p.\mathsf{child}[q]$ becomes `false` in finite time.

If $\gamma(p).\mathsf{child}[q] = \mathtt{true}$, p sends a message $\langle \text{PAR} \rangle$ to q in at most 3 rounds (Line 33). At most 3 rounds later, q receives this message (Line 14) and sends back a message $\langle \text{NOCHILD} \rangle$ since $q.\mathsf{par} \neq p$ (Line 15). When p receives this message at most 3 rounds later (Line 16), it sets $p.\mathsf{child}[q]$ to `false` (Line 21). Since $q.\mathsf{par}$ remains different than p afterwards, q can never send a feedback containing $\langle \mathtt{true} \rangle$ or a message $\langle \text{CHILD}, * \rangle$ to p. Thus, p never sets $p.\mathsf{child}[q]$ to `true` again. Hence, in at most 9 rounds after γ, $\forall q \in \mathcal{N}_p$, $p.\mathsf{child}[q] = \mathtt{false}$ so $Passive(p)$ continuously holds. In at most 3 additional rounds, p gets status PASS and leaves the tree by setting $p.\mathsf{par}$ to $\bot$ (Lines 25-28). So the height of the tree that contained p in γ decreases in at most 12 rounds.

By repeating this argument, eventually every tree is composed of only one process of status PASS. In particular, the initiator signals quiescence when it gets status PASS (Line 28). Since the height of the trees is at most $n-1$, the initiator signals quiescence at most $12n$ rounds after γ. □

Let us define a last property on $p \in V$ and its neighbors $q \in \mathcal{N}_p$.

Definition 6. *Let $p \in V$ and $q \in \mathcal{N}_p$.*
$C_8(p,q)$. $p.\mathsf{status} = \text{PASS} \wedge q.\mathsf{status} \neq \text{ERR} \Rightarrow q.\mathsf{par} \neq p$

Lemma 3. *Let $p \in V$ and $q \in \mathcal{N}_p$.*

1. *Let $\gamma \mapsto \gamma'$ s.t. no tainted messages are in the channels. If $C_5(p,q)$ and $C_8(p,q)$ hold in γ then $C_8(p,q)$ holds in γ'.*
2. *In at most 33 rounds, $C_8(p,q)$ holds.*

Proof.

1. Let $\gamma \mapsto \gamma'$ s.t. there is no tainted messages in the channels, and $C_5(p,q)$ and $C_8(p,q)$ hold in γ. A process with status ERR cannot change it. Assume $\gamma'(p).\mathsf{status} = \text{PASS}$ and $\gamma'(q).\mathsf{status} \neq \text{ERR}$.
   (a) If $\gamma(p).\mathsf{status} = \text{PASS}$ then, by C_8, $\gamma(q).\mathsf{par} \neq p$. Assume q sets $q.\mathsf{par}$ to p during $\gamma \mapsto \gamma'$, it receives a packet $pckt$ from p (Line 4). However, when p initiates the PIF to send $pckt$ (Line 32), $p.\mathsf{pckt}[q] \neq \bot$ and $p.\mathsf{status} = \text{ACT}$. Moreover, p cannot get status PASS without receiving a feedback from q, which must happen after $\gamma \mapsto \gamma'$ in which $pckt$ is assumed to be received by q (Proposition 2), a contradiction. Hence $\gamma'(q).\mathsf{par} \neq p$.
   (b) If p gets status PASS during $\gamma \mapsto \gamma'$, $\gamma(p).\mathsf{pckt}[q] = \bot$ and $\gamma(p).\mathsf{child}[q] = \mathtt{false}$. Thus, by the contrapositive of $C_5(p,q)$, $\gamma(q).\mathsf{par} \neq p$. Similarly to case 1a, q cannot set $q.\mathsf{par}$ to p.
2. Let γ_T be the first configuration after 27 rounds. By Corollary 1, no tainted messages are in the channels. If $\gamma_T(p).\mathsf{status} = \text{ACT}$ but $\gamma_T(q).\mathsf{par} = p$ then q sends a message $\langle \text{CHILD}, * \rangle$ to p in at most 3 rounds (Line 34). When p receives this message at most 3 rounds later, it gets status ERR (Line 20). □

Definition 7. *An initial configuration γ_0 is* regular *if:*

R_1. $\forall p \in V$, $\gamma_0(p).\mathsf{init} \Rightarrow (\gamma_0(p).\mathsf{dist} = 0 \land \gamma_0(p).\mathsf{par} = \bot)$
R_2. $\forall p \in V$, $\neg\gamma_0(p).\mathsf{init} \Rightarrow (\gamma_0(p).\mathsf{par} = \bot \land \forall q \in \mathcal{N}_p, \neg\gamma_0(p).\mathsf{child}[q])$
R_3. $\forall p \in V$, $\gamma_0(p).\mathsf{status} = \text{PASS} \land \forall q \in \mathcal{N}_p, \gamma_0(p).\mathsf{pckt}[q] = \bot$
R_4. *There is no messages in the channels in* γ_0.

By Definition 7 and Lemmas 1, 2, and 3, we have the following corollary.

Corollary 3. *In any configuration γ of an execution starting from a regular initial configuration γ_0 such that every execution of $\mathcal{A}$ starting from $\gamma_0(\mathcal{A})$ is a diffusing computation, $\forall p \in V$, $LCorrect(p)$ holds and $\mathcal{G}_{ex}$ is a forest in γ. Moreover, $\forall p \in V$, $\forall q \in \mathcal{N}_p$, $C_5(p,q)$, $C_6(p,q)$, $C_7(p,q)$, and $C_8(p,q)$ hold in γ.*

Theorem 2 (Soundness)**.** *If $SigQuiet()$ is called, either the execution of $\mathcal{A}$ actually reached quiescence or was not a diffusing computation, or the initial configuration of $\mathcal{Q}$ was not regular.*

Proof. We prove this theorem by the contrapositive. Assume the execution of $\mathcal{A}$ never reaches quiescence (*i.e.*, there is always some enabled process in $\mathcal{A}$ or a process that needs to send or receive a packet of $\mathcal{Q}$) and is a diffusing computation, and the initial configuration of $\mathcal{Q}$ is regular. Assume by contradiction that $\mathcal{Q}$ signals quiescence.

By hypothesis, quiescence is signaled before any error signal. Moreover, a process gets status ERR before signaling an error and the initial status of every process is PASS. So no process ever gets status ERR. Let r be the only initiator and thus the process that signaled quiescence. Let $\gamma_i \mapsto \gamma_{i+1}$ be the step where r calls $SigQuiet()$ (Line 28). Thus, $Passive(r)$ holds in γ_i. Since the execution of $\mathcal{A}$ has not reached quiescence, there are three cases:

1. $\exists p \in V$ s.t. $\neg Quiet(p)$ holds in γ_i. Thus, $\neg Passive(p)$ holds in γ_i.
2. A packet of $\mathcal{A}$ must be sent, *i.e.*, $\exists p \in V$, $\exists q \in \mathcal{N}_p$, s.t. $\gamma_i(p).\mathsf{pckt}[q] \neq \bot$. Thus, $\neg Passive(p)$ holds in γ_i.
3. A PIF has been initiated by some process p to send a packet *pckt* to some neighbor q, yet q did not received *pckt* yet. In this latter case, when p initiated the PIF (Line 32), $p.\mathsf{pckt}[q] = pckt \neq \bot$. Moreover, p cannot set $p.\mathsf{pckt}$ to $\bot$ until p receives a feedback from q. However, q cannot send such a feedback before receiving *pckt*. Thus, $\gamma_i(p).\mathsf{pckt}[q] \neq \bot$ and $\neg Passive(p)$ holds in γ_i.

In those three cases, $\exists p \in V$ s.t. $\neg Passive(p)$ holds in γ_i. Notice that $p \neq r$.

The initial configuration is regular, $\neg Passive(p)$ holds in γ_i, and p is not the initiator. Thus, p received a first packet from some neighbor x (Line 1) at some point in the execution. Again, since the initial configuration is regular, p sets $p.\mathsf{par}$ to x and gets status ACT. Consider the last time p changes its status from PASS to ACT (say in $\gamma_j \mapsto \gamma_{j+1}$) before γ_i, *i.e.*, $j < i$, $\gamma_j(p).\mathsf{status} = \text{PASS}$, and $\forall j' \in \{j+1,\ldots,i\}$, $\gamma_{j'}(p).\mathsf{status} = \text{ACT}$. Again, in order to get status ACT, p had received a packet from some neighbor y (Line 1) in $\gamma_j \mapsto \gamma_{j+1}$. Hence, $\gamma_{j+1}(p).\mathsf{par} = y$. Process p cannot change the value of $p.\mathsf{par}$ without getting status PASS (Line 27). Hence, $\gamma_i(p).\mathsf{par} = y \neq \bot$.

By Corollary 3, Property C_6, and since $\mathcal{G}_{ex}$ is a forest, there is a sequence of distinct processes $seq = p_0, p_1, \ldots, p_k$, $0 < k \geq n$, such that $p_0 = p$, $\forall i \in \{0,\ldots,k-1\}$, $\gamma(p_i).\mathsf{par} = p_{i+1}$, and $p_k = r$. Now, by recursively applying $C_5(p_i, p_{i+1})$ for $i \in \{0,\ldots,k-1\}$, we have $\gamma(p_{i+1}).\mathsf{pckt}[p_i] \neq \bot \lor \gamma(p_{i+1}).\mathsf{child}[p_i]$. Thus, $\neg Passive(p_{i+1})$ holds in γ_i. In particular, $\neg Passive(r)$ holds in γ_i, a contradiction. ☐

Theorem 3 (Relevance)**.** *If the initial configuration of $\mathcal{Q}$ is regular and the execution of $\mathcal{A}$ is a diffusing computation, no process ever calls $SigError()$.*

Proof. Let $p \in V$. By Corollary 3, $LCorrect(p)$ holds throughout the execution so p cannot call $SigError()$ executing Lines 24 without already being in status ERR. Initially, $p.\mathsf{status} \neq$ ERR

since the initial configuration is regular. Except by executing Lines 24, p can get status $\textsc{Err}$ when receiving a feedback (Line 12) or a message $\langle\textsc{Child}, *\rangle$ (Line 20). In both cases, p immediately calls $SigError()$ (Lines 12 and 20, respectively). Thus, there are two cases.

1. If p calls $SigError()$ executing Line 12, it has just received a feedback $fbck$ containing $\langle\texttt{true}\rangle$ from some $q \in \mathcal{N}_p$ while $p.\mathsf{status} = \textsc{Pass}$ or $\neg p.\mathsf{init} \wedge p.\mathsf{par} = \bot$. When q sent $fbck$ (say in $\gamma \mapsto \gamma'$), $q.\mathsf{par} = p$. By Corollary 3, $C_5(q,p)$, $C_6(q,p)$, and the contrapositive of $C_8(q,p)$, $\gamma(p).\mathsf{status} = \textsc{Act}$, $\gamma(p).\mathsf{pckt}[q] \neq \bot \vee \gamma(p).\mathsf{child}[q]$, and $\gamma(p).\mathsf{init} \vee \gamma(p).\mathsf{par} \neq \bot$. So p cannot get status $\textsc{Pass}$ or change the value of its par-variable before receiving $fbck$ (Proposition 2), a contradiction.
2. If p calls $SigError()$ executing Line 20, it received some message $msg = \langle\textsc{Child}, dist\rangle$ from a neighbor $q \in \mathcal{N}_p$ while (a) $p.\mathsf{status} = \textsc{Pass}$, or (b) $\neg p.\mathsf{init} \wedge\ p.\mathsf{par} = \bot$, or (c) $dist \neq p.\mathsf{dist} + 1$. When q sent msg (say in $\gamma \mapsto \gamma'$), $q.\mathsf{par} = p$ and $q.\mathsf{dist} = dist$. Thus, situations (a) and (b) are similar to case 1. Now, consider situation (c). By Corollary 3, $C_5(q,p)$, $C_7(q,p)$, and the contrapositive of $C_8(q,p)$, we have $\gamma(p).\mathsf{status} = \textsc{Act}$, $\gamma(p).\mathsf{pckt}[q] \neq \bot \vee \gamma(p).\mathsf{child}[q]$, and $\gamma(p).\mathsf{dist} = \gamma(q).\mathsf{dist} + 1 = dist + 1$. Process p cannot change its distance without getting status $\textsc{Pass}$ and so, without $q.\mathsf{par}$ becoming different than p. If $q.\mathsf{par}$ remains equal to p in between γ and the reception of m by p, p never changes its distance, a contradiction. Otherwise, $q.\mathsf{par}$ becomes different than p at some point between γ and the reception of m by p. To allow p to get status $\textsc{Pass}$, q must send a message $msg' = \langle\textsc{NoChild}\rangle$ or a feedback $fbck$ containing $\langle\texttt{false}\rangle$ to p after changing its par-variable, *i.e.*, after sending msg. Moreover, p must receive msg' or $fbck$ before msg even if the channels are FIFO, a contradiction. □

4.4 Message Complexity

Finally, we study the message complexity of $\mathcal{Q}$ in *k-synchronous* execution. We adapt the definition of k-synchronous executions from [15] to message-passing systems.

Definition 8. *An execution Γ is* k-synchronous *if the following conditions hold.*

a). The difference of speed between the slowest and fastest messages is at most k. More precisely, let $\gamma_{s_m} \mapsto \gamma_{s_m+1}$ and $\gamma_{r_m} \mapsto \gamma_{r_m+1}$ be the steps during which the message m is sent and received, respectively. For every pair of messages m, m' sent during Γ, $(r_m - s_m) \leq k(r_{m'} - s_{m'})$.

b). The difference of speed between the slowest and fastest processes is at most k. More precisely, for any $\Gamma_0, \Gamma_1, \Gamma'$ such that $\Gamma = \Gamma_0\Gamma_1\Gamma'$, and for any two processes p and q, if q triggers at least $k+1$ timeout during Γ_1 then p triggers at least one timeout during Γ_1.

Theorem 4 (Message Complexity)**.** *If Γ is k-synchronous and any execution of $\mathcal{A}$ starting from $\gamma_0(\mathcal{A})$ reaches quiescence, then $O(k(m + n(t_{\mathcal{A}} + n) + M_{\mathcal{A}}))$ messages are sent before some process calls $SigQuiet()$ or $SigError()$, where n is the number of processes, m is the number of edges, $t_{\mathcal{A}}$ (respectively, $M_{\mathcal{A}}$) is the maximum number of rounds (respectively, of exchanged messages in $\mathcal{A}$) for $\mathcal{A}$ to reach quiescence from $\gamma_0(\mathcal{A})$.*

Proof. Let p be a process. Γ is k-synchronous and a process sends at most one message per neighbor at each activation, so p sends at most k messages per neighbor during one round. By Corollary 2, $\mathcal{G}_{ex}$ becomes a forest in at most 33 rounds. During these 33 rounds, up to $O(k\,m)$ messages are exchanged. The remaining of the computation before a process signals quiescence or an error lasts $O(t_{\mathcal{A}} + n)$ rounds. During this part of the computation, messages $\langle\textsc{Child}, *\rangle$ and $\langle\textsc{Par}\rangle$ are only exchanged along the trees, *i.e.*, a total of $O(k(n-1)(t_{\mathcal{A}} + n))$ messages. Moreover, the transmission of a packet and the corresponding feedback using $\mathcal{PIF}$ lasts 27 rounds during which the emitter and the recipient of the packet exchange $\mathcal{PIF}$ messages

(Proposition 1), *i.e.*, a total of $O(k\ M_{\mathcal{A}})$ messages. Hence $O(k(m + (n-1)(t_{\mathcal{A}} + n) + M_{\mathcal{A}}))$ messages are exchanged before a signal. □

Remark 1. *Notice that, if the initial configuration of $\mathcal{Q}$ is regular and the execution of $\mathcal{A}$ is a diffusing computation, $\mathcal{G}_{ex}$ is always a forest. Thus, in this case, $O(k(n(t_{\mathcal{A}} + n) + M_{\mathcal{A}}))$ messages are exchanged before a signal.*

5 Discussion and Future Work

We proposed the first self-stabilizing and snap-stabilizing ongoing quiescence detection algorithm $\mathcal{Q}$. This algorithm works for diffusing computations. One can use $\mathcal{Q}$ to detect termination before safely re-starting a task or starting a new one, for example to transform a non self-stabilizing algorithm into a self-stabilizing one. This transformer is more efficient than the Awerbuch and Varghese transformation. For example, let consider a non self-stabilizing algorithm $\mathcal{A}$ whose time and message complexity are x and y, respectively. If no faults hit the system (in particular, the initial configuration of $\mathcal{Q}$ is regular), the transformer using $\mathcal{Q}$ requires $O(k(n\ x + n^2 + y))$ messages before detecting the termination of $\mathcal{A}$, if the execution is k-synchronous. In a similar context, the transformer of Awerbuch and Varghese uses m messages per time unit for the synchronizer. Hence, $\Omega(k\ m\ x + y)$ messages are exchanged. (*N.B.*, k is a constant.)

In addition to the improved performances, the proposed transformer has other advantages over the method with a synchronizer. Indeed, it does not require to know the time complexity of $\mathcal{A}$ contrary to the previous methods that use this bound to know when the execution of $\mathcal{A}$ is terminated. Moreover, the time complexity of $\mathcal{A}$ is an upper bound on the time before $\mathcal{A}$ terminates, but an execution of $\mathcal{A}$ can actually terminate (far) earlier. In this case, our method stabilizes faster since it detects termination when it happens. These advantages hold only if the execution of $\mathcal{A}$ terminates. Otherwise, it requires a termination enforcement method like the one proposed above.

A natural open problem is the generalization of this quiescence detection algorithm to non-diffusing computations.

References

[1] Yehuda Afek, Shay Kutten, and Moti Yung. The local detection paradigm and its application to self-stabilization. *Theor. Comput. Sci.*, 186(1-2):199–229, 1997.

[2] Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. Time optimal self-stabilizing synchronization. In *STOC'93*, pages 652–661, 1993.

[3] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction (extended abstract). In *FOCS'91*, pages 268–277, 1991.

[4] Baruch Awerbuch, Boaz Patt-Shamir, George Varghese, and Shlomi Dolev. Self-stabilization by local checking and global reset. In *WDAG'94*, pages 326–339, 1994.

[5] Baruch Awerbuch and George Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *FOCS'91*, pages 258–267, 1991.

[6] Christian Boulinier, Franck Petit, and Vincent Villain. When graph theory helps self-stabilization. In *PODC'04*, pages 150–159, 2004.

[7] Alain Bui, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. State-optimal snap-stabilizing PIF in tree networks. In *WSS'99*, pages 78–85, 1999.

[8] K. Mani Chandy and Jayadev Misra. An example of stepwise refinement of distributed programs: Quiescence detection. *ACM TOPLAS*, 8(3):326–343, 1986.

[9] Alain Cournier, Ajoy Kumar Datta, Stéphane Devismes, Franck Petit, and Vincent Villain. The expressive power of snap-stabilization. *Theor. Comput. Sci.*, 626:40–66, 2016.

[10] Sylvie Delaët, Stéphane Devismes, Mikhail Nesterenko, and Sébastien Tixeuil. Snap-stabilization in message-passing systems. *J. Parallel Distrib. Comput.*, 70(12):1220–1230, 2010.

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

[12] Edsger W. Dijkstra and Carel S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

[13] Nissim Francez. Distributed termination. *ACM TOPLAS*, 2(1):42–55, 1980.

[14] Nissim Francez, Michael Rodeh, and Michel Sintzoff. Distributed termination with interval assertions. In *Formalization of Programming Concepts*, pages 280–291, 1981.

[15] Danny Hendler and Shay Kutten. Bounded-wait combining: constructing robust and high-throughput shared objects. *Distributed Computing*, 21(6):405–431, 2009.

[16] Shmuel Katz and Kenneth J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7(1):17–26, 1993.

[17] Amos Korman, Shay Kutten, and Toshimitsu Masuzawa. Fast and compact self-stabilizing verification, computation, and fault detection of an MST. In *PODC'11*, pages 311–320, 2011.

[18] Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Computing*, 22(4):215–233, 2010.

[19] Florence Levé, Khaled Mohamed, and Vincent Villain. Snap-stabilizing PIF on arbitrary connected networks in message passing model. In *SSS'16*, pages 281–297, 2016.

[20] Jeff Matocha and Tracy Camp. A taxonomy of distributed termination detection algorithms. *Journal of Systems and Software*, 43(3):207–221, 1998.

[21] George Varghese. Self-stabilization by counter flushing. *SIAM J. Comput.*, 30(2):486–510, 2000.

A Appendix

In this appendix, we detail the proof of Lemmas 1 and 2.

Lemma 1. *Let $p \in V$. If the execution of $\mathcal{A}$ is a diffusing computation, we have the following results.*

1. *Let $\gamma \mapsto \gamma'$. If $LCorrect(p)$ holds in γ then $LCorrect(p)$ holds in γ'.*
2. *In at most 3 rounds, $LCorrect(p)$ holds.*

Proof.

1. Let $\gamma \mapsto \gamma'$ s.t. $LCorrect(p)$ holds in γ. A process with status Err cannot change its status. Now, if $\gamma(p).\mathsf{status} \neq$ Err:

   C_1. An initiator cannot change its parent or distance. Moreover a non-initiator cannot become initiator. Thus, $C_1(p)$ holds in γ'.

   C_2. When a process p sets $p.\mathsf{child}[q]$ to `true` (Lines 11 or 19), p is an initiator or has a parent. Again, an initiator cannot become non-initiator. Moreover, a non-initiator cannot set its variable par to $\perp$ if it has children. Thus, $C_2(p)$ holds in γ'.

   C_3. By definition of a diffusing computation, p can only send a packet of $\mathcal{A}$, *i.e.*, $p.\mathsf{pckt}[q] \neq \perp$, if p is the initiator or already received a packet (Line 1). In the latter case, p set $p.\mathsf{par}$ to a value different than $\perp$ (Line 4). Thus, $C_3(p)$ holds in γ'.

   C_4. It is not possible for p to get status Pass without setting $p.\mathsf{par}$ to $\perp$ (Lines 25-28). Moreover, p cannot set $p.\mathsf{par}$ to a value different than $\perp$ without getting status Act (Line 4). Thus, $C_4(p)$ holds in γ'.

2. If $\neg LCorrect(p)$, p gets status Err in at most 3 rounds (Lines 24). □

Lemma 2. *Let $p \in V$ and $q \in \mathcal{N}_p$.*

1. *Let $\gamma \mapsto \gamma'$ s.t. no tainted messages are in the channels. If $C_5(p,q)$, $C_6(p,q)$, and $C_7(p,q)$ hold in γ then $C_5(p,q)$, $C_6(p,q)$, and $C_7(p,q)$ hold in γ'.*
2. *In at most 33 rounds, $C_5(p,q)$, $C_6(p,q)$, and $C_7(p,q)$ hold.*

Proof.

1. Let $\gamma \mapsto \gamma'$ s.t. no tainted messages are in the channels and $C_5(p,q)$, $C_6(p,q)$, and $C_7(p,q)$ hold in γ. A process with status Err cannot change its status. Assume $\gamma'(p).\mathsf{status} \neq$ Err, $\gamma'(q).\mathsf{status} \neq$ Err, and $\gamma'(p).\mathsf{par} = q$. Then, there are two cases: either $\gamma(p).\mathsf{par} = \gamma'(p).\mathsf{par} = q$, or $\gamma(p).\mathsf{par} \neq q$ and p sets $p.\mathsf{par}$ to q during $\gamma \mapsto \gamma'$.

   (a) If $\gamma(p).\mathsf{par} = q$ then, by $C_5(p,q)$, $\gamma(q).\mathsf{child}[p] = \mathtt{true}$ or $\gamma(q).\mathsf{pckt}[p] \neq \perp$. If q is the initiator it cannot become non-initiator. Moreover, $q.\mathsf{par}$ and $q.\mathsf{dist}$ cannot change as long as $(q.\mathsf{child}[p] \vee q.\mathsf{pckt}[p] \neq \perp)$. Now, let us show that $(q.\mathsf{child}[p] \vee q.\mathsf{pckt}[p])$ continuously holds between γ and γ'.

   If $\gamma(q).\mathsf{pckt}[p] \neq \perp$, q can only set $q.\mathsf{pckt}[p]$ to a $\perp$ if it receives a feedback from p (Line 7). However, since there is no tainted messages in the channels, this feedback was sent by p after it receives some packet from q (Proposition 2). Hence, the value inside the feedback can only be `true` as long as $p.\mathsf{par} = q$. In this case, $\gamma'(q).\mathsf{child}[p] = \mathtt{true}$.

   Now, if $\gamma(q).\mathsf{child}[p] = \mathtt{true}$, q can only set $q.\mathsf{child}[p]$ to `false` if it receives a feedback from p with value $\langle\mathtt{false}\rangle$ or a message $\langle$NoChild$\rangle$ from p. As long as $p.\mathsf{par} = q$, p cannot send such messages, so $\gamma'(q).\mathsf{child}[p] = \mathtt{true}$. Thus, $C_5(p,q)$ and $C_6(p,q)$ hold in γ'.

   Finally, p and q cannot change their distance without changing their parent, thus $C_7(p,q)$ also hold in γ'.

(b) If p sets $p.\mathsf{par}$ to q during $\gamma \mapsto \gamma'$, it receives a packet $m = \langle pckt, dist \rangle$ from q (Lines 1-4) and sets $p.\mathsf{dist}$ to $dist + 1$. Since no tainted messages are in the channels, the PIF used to send m has been initiated by q (Proposition 2). When q sent m (Line 32), $q.\mathsf{pckt}[p] \neq \bot$, $q.\mathsf{dist} = dist$, and $q.\mathsf{init} \vee q.\mathsf{par} \neq \bot$. Similarly to case 1a, this hold until q receives a feedback from p containing $\langle \mathtt{false} \rangle$. Again, q cannot receive such a feedback before p received m (Proposition 2). Hence, $C_5(p, q)$, $C_6(p, q)$, and $C_7(p, q)$ hold in γ'.

2. Let γ_T be the first configuration after 27 rounds. By Corollary 1, no tainted messages are in the channels. If $\gamma_T(p).\mathsf{status} \neq \textsc{Err}$, $\gamma_T(q).\mathsf{status} \neq \textsc{Err}$, and $\gamma_T(p).\mathsf{par} = q$ then, in at most 3 rounds, p sends a message $m = \langle \textsc{Child}, dist \rangle$ where $dist = p.\mathsf{dist}$ (Line 34). (Notice that p cannot change its distance as long as $p.\mathsf{par} = q$.) When q receives m at most 3 rounds later (Line 16), q gets status Err if $\neg q.\mathsf{init} \wedge q.\mathsf{par} = \bot$ or if $dist \neq q.\mathsf{dist} + 1$ (Line 20). Otherwise, q sets $q.\mathsf{child}[p]$ to $\mathtt{true}$ (Line 19). Hence, $C_5(p, q)$, $C_6(p, q)$, and $C_7(p, q)$ hold after 27 rounds. $\square$