The expressive power of snap-stabilization

Alain Cournier, Ajoy K. Datta, Stéphane Devismes, Franck Petit, Vincent Villain

▶ **To cite this version:**

Alain Cournier, Ajoy K. Datta, Stéphane Devismes, Franck Petit, Vincent Villain. The expressive power of snap-stabilization. Theoretical Computer Science, 2016, 626, pp.40-66. 10.1016/j.tcs.2016.01.036 . hal-01292988

HAL Id: hal-01292988

https://hal.sorbonne-universite.fr/hal-01292988

Submitted on 24 Mar 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

The Expressive Power of Snap-Stabilization*

Alain Cournier[1] Ajoy K. Datta[2] Stéphane Devismes[3] Franck Petit[4]
Vincent Villain[1]

March 23, 2016

[1] MIS, Université de Picardie Jules Verne, Amiens (France)
firstname.lastname@u-picardie.fr
[2] Department of Computer Science, University of Nevada Las Vegas (USA)
ajoy.datta@unlv.edu
[3] VERIMAG, UMR 5104, Université de Grenoble (France)
stephane.devismes@imag.fr
[4] LIP6, Université Pierre et Marie Curie, Paris (France)
franck.petit@lip6.fr

Abstract

A *snap-stabilizing* algorithm, regardless of the initial configuration of the system, guarantees that it always behaves according to its specification. We consider here the locally shared memory model. In this model, we propose the first snap-stabilizing Propagation of Information with Feedback (PIF) algorithm for rooted networks of arbitrary connected topology which is proven assuming the distributed unfair daemon. Then, we use the proposed PIF algorithm as a key module in designing snap-stabilizing solutions for some fundamental problems in distributed systems, such as Leader Election, Reset, Snapshot, and Termination Detection. Finally, we show that in the locally shared memory model, snap-stabilization is as expressive as self-stabilization by designing a universal transformer to provide a snap-stabilizing version of any algorithm that can be self-stabilized with the transformer of Katz and Perry (Distributed Computing, 1993). Since by definition a snap-stabilizing algorithm is also self-stabilizing, self- and snap-stabilization have the same expressiveness in the locally shared memory model.

Keywords: Fault-tolerance, snap-stabilization, self-stabilization, propagation of information with feedback, leader election, reset, snapshot.

*This paper is an extended version of preliminary results [1, 2].

Contents

1 Introduction **3**

2 Preliminaries **7**
2.1 Distributed Systems 7
2.2 Computational Model 7

3 Snap-Stabilization **8**
3.1 Normal vs. Legitimate Configurations 8
3.2 Formal Definition of Snap-Stabilization 9
3.3 Delay vs. Stabilization Time 9

4 Snap-Stabilizing PIF **10**
4.1 The Algorithm 11
4.1.1 PIF Part 11
4.1.2 Question Part 12
4.1.3 Correction Part 13
4.2 Proof of Snap-Stabilization of Algorithm $\mathcal{PIF}$ 14
4.2.1 Definitions 14
4.2.2 Proof assuming a Weakly Fair Daemon 15
4.2.3 Proof assuming an Unfair Daemon 20
4.3 Complexity Analysis 23

5 Other Key Algorithms **24**
5.1 Snap-Stabilizing Leader Election 24
5.2 Snap-Stabilizing Reset 24
5.3 Snap-Stabilizing Snapshot 24
5.4 Snap-Stabilizing Termination Detection 25

6 Transformer **25**
6.1 Single Initiator Algorithm 25
6.2 Multi-Initiator Algorithm 27

7 Conclusion **31**

1 Introduction

Context Modern distributed systems are made of a large number of interconnected processes. Increasing the number of components (processes or links) in a distributed system means increasing the probability that some of these components fail during the execution of a distributed algorithm. Moreover, due to the large scale, human intervention to quickly repair failed components is also not possible. So, in this context, fault-tolerance, *e.g.*, the ability of a distributed algorithm to endure faults, is mandatory.

We consider a particular type of faults, called the *transient faults*. A transient fault occurs at an unpredictable time but does not result in a permanent hardware damage. Moreover, as opposed to intermittent faults, the frequency of transient faults is considered to be low. Consequently, network components affected by transient faults temporarily deviate from their specifications, *e.g.*, some messages in a link may be lost, reordered, duplicated, or corrupted. As a result, a transient fault affects the state of the component in which it occurs. Hence, after a finite number of transient faults, the configuration of a distributed system can be arbitrary, *i.e.*, process memories can be corrupted and communication links may contain corrupted messages.

In 1974, Dijkstra [3] proposed a general paradigm called *self-stabilization* to enable the design of distributed systems tolerating *any* finite number of transient faults. Consider the first configuration after all transient faults cease. This configuration is arbitrary, but no other transient faults will ever occur from this configuration. By abuse of language, this configuration is referred to as *arbitrary initial configuration* of the system in the literature. Then, a self-stabilizing algorithm (provided that faults have not corrupted its code) guarantees that starting from an arbitrary initial configuration, the system recovers *within finite time*, without any external intervention, to a configuration from which its specification is (always) satisfied. Thus, self-stabilization makes no hypotheses on the nature or extent of transient faults that could hit the system, and the system recovers from the effects of those faults in a unified manner. Such versatility comes at a price, *e.g.*, after transient faults cease, there is a finite period of time, called the *stabilization phase*, during which the safety properties of the system may be violated. Hence, self-stabilizing algorithms are mainly compared according to their stabilization time, the maximum duration of the stabilization phase.

Several approaches have been introduced to offer more stringent guarantees than simple eventual recovery, *e.g.*, *fault-containment* [4], *superstabilization* [5], and *snap-stabilization* [6, 7]. We focus here on the concept of *snap-stabilization*, introduced by Datta *et al* in 1999 [6, 7]. Snap-stabilization is a stronger form of self-stabilization, as after transient faults cease, a snap-stabilizing system *immediately* resumes correct behavior, without any external intervention, provided the faults have not corrupted its code. More formally, starting from an arbitrary initial configuration (*i.e.*, the first configuration after the end of the faults), a snap-stabilizing system (always) satisfies its specification. Hence, by definition, snap-stabilizing algorithms are self-stabilizing algorithms whose stabilization time is null.

To illustrate the advantage of snap-stabilization compared with self-stabilization, we now consider the fundamental problem of *termination detection*. In this problem, any process p can be requested (by the application layer) to detect if some distributed algorithm $\mathcal{X}$ has terminated. More precisely, upon a request a process should initiate a query to know if $\mathcal{X}$ has terminated, and when p delivers an answer "yes" (resp., "no"), $\mathcal{X}$ "has terminated" (resp., "may not have terminated"). Let $\mathcal{A}_{\texttt{self}}$ (resp., $\mathcal{A}_{\texttt{snap}}$) be a self-stabilizing (resp., snap-stabilizing) algorithm for detecting the termination of some distributed algorithm $\mathcal{X}$. Let p be any process. If $\mathcal{A}_{\texttt{self}}$ starts from an arbitrary initial configuration and $\mathcal{X}$ eventually terminates, then all we know is that eventually only "yes" answers will be computed for all p's queries, and these answers will truly indicate that $\mathcal{X}$ did terminate. However, during the stabilization phase, it is possible that p delivers "yes" answers while $\mathcal{X}$ actually has not terminated. In other words, $\mathcal{A}_{\texttt{self}}$ can compute false (or unsafe) answers several (but a finite number of) times before finally computing true/correct answers. In contrast, using $\mathcal{A}_{\texttt{snap}}$, starting from an arbitrary initial configuration, the very first answer delivered by p to any initiated query can be trusted to be a true answer.

It is important to note that snap-stabilizing systems are not insensitive to transient faults. For example, using $\mathcal{A}_{\texttt{snap}}$, if a transient fault occurs between an initiated query and its associated answer, then the answer may not be correct, *i.e.*, a process may deliver "yes" while $\mathcal{X}$ actually has not terminated. However, every answer returned to any query initiated after the end of faults will be correct. In contrast, $\mathcal{A}_{\texttt{self}}$ just guarantees that only a finite, yet generally unbounded, number of false answers will be returned after faults cease.

Related Work Since the seminal work of Bui *et al* [6, 7], many snap-stabilizing solutions, dedicated to various problems and handling different network topologies, have been proposed. Most notably, numerous papers on snap-stabilization deal with the *Propagation of Information with Feedback* (PIF) and *depth-first token circulation* problems. The former has been addressed in rooted tree networks [8, 9] and arbitrary connected rooted networks [10, 11, 12].

Similarly, snap-stabilizing algorithms that solve the depth-first token circulation are given in rooted trees [13], arbitrary connected rooted networks [12, 14], and arbitrary connected, rooted, and identified networks [15].

Snap-stabilization has been also addressed to solve other problems in various contexts, *e.g.*, computing binary search trees [16], cutset detection [17], neighborhood synchronization in trees [18], global synchronization in trees [19], committee coordination [20], computing prefix trees in Peer-to-peer systems [21], and linear message forwarding [22, 23, 24]. Notice that [24] constitutes the first attempt to deal with snap-stabilization in the context of dynamic networks.

Most of the above work is designed in the locally shared memory model. Only a few papers deal with snap-stabilization in message-passing systems [25, 26, 27]. In [25], the authors sketch a snap-stabilizing snapshot algorithm for oriented tree networks. The algorithms given in [26] deals with PIF and mutual exclusion in complete networks. Mohamed *et al* proposed a snap-stabilizing PIF algorithm for unoriented tree networks in [27].

Motivation With such a diversity, a natural question arises: "*How expressive is snap-stabilization?*" This is the main focus of this paper. The expressiveness of *self-stabilization* has been already investigated by Katz and Perry [28], as they proposed a self-stabilizing snapshot algorithm working in the message-passing model and used it to transform most of the non-self-stabilizing algorithms into self-stabilizing ones. Formally, Katz and Perry [28] consider as possible inputs of their transformer all algorithms solving problems that can be defined by a *suffix-closed* specification.[1] In the same paper, they also show that this condition is necessary.

In this paper, we show that for any identified distributed system in the locally shared memory model, *self-* and *snap-stabilization* have the same *expressiveness*, meaning that any problem for which there exists a self-stabilizing solution, there also exists a snap-stabilizing one, and *vice versa*. By definition, snap-stabilizing algorithms are self-stabilizing algorithms whose stabilization time is null. So, the only interesting question is to demonstrate the first part of our assertion. This result may appear surprising and even contradictory to the well-known lower bounds on time complexity in the self-stabilization literature (*e.g.*, $\Omega(D)$ where D is the diameter of the network [29]). We first aim at convincing the reader that this apparent contradiction comes from a misunderstanding between the notion of *problem* and the notion of *specification* of a problem. Once we clarify that misconception, we hope that the readers will appreciate the interest of our approach for stabilization and the very desirable practical property we can get from it.

The notion of problem is not easy to define, and it could be impossible to formally define it *a priori*, like it is impossible for the well-known notion of set for example. The notion of specification could be easier to capture: a specification is a set of properties or predicates that an algorithm must satisfy. We now build the framework in which we will be able to define a problem. Classically, the correctness of a *non fault-tolerant* algorithm is established by observing the system from a pre-defined configuration, called *initial configuration*, from which the system is supposed to start. Moreover, the system is supposed to not suffer from any fault all along its execution. In the following, we will say that such systems ensure a *safe environment*. We can now formally define the notion of problem as *an equivalence class of specifications in a safe environment* where the equivalence relation is as follows: for every two specifications $\mathcal{SP}_1$ and $\mathcal{SP}_2$, $\mathcal{SP}_1$ and $\mathcal{SP}_2$ are equivalent if and only if any algorithm satisfying $\mathcal{SP}_1$ in a safe environment also satisfies $\mathcal{SP}_2$ in a safe environment, and *vice versa*.

Of course, if the environment changes and considering an equivalence relation in this new environment, some specifications which were previously equivalent may not be equivalent anymore. For example, a specification may or may not make the distinction between faulty and correct processes. As a consequence, one can get two different specifications, one for each assumption. More precisely, consider the well-known (binary) *consensus* problem, which considers that each process has an input value 0 or 1, and should decide unanimously (*i.e.* output) a value, either 0 or 1. Let us recall two consensus specifications, thereafter denoted $\mathcal{SP}_1$ and $\mathcal{SP}_2$, which appear in the literature. $\mathcal{SP}_1$ [30] can be used in the context of safe environments, while $\mathcal{SP}_2$ [31] applies to systems subject to process crashes. Both Specifications $\mathcal{SP}_1$ and $\mathcal{SP}_2$ include the two following requirements:

Integrity: Every process decides at most once.

Validity: If a process decides a value v, then v was proposed by some process.

However, $\mathcal{SP}_1$ and $\mathcal{SP}_2$ differ by two different notions of termination and agreement. In Specification $\mathcal{SP}_1$, we have:

Uniform Termination: Every process eventually decides some value.

[1] A specification $\mathcal{S}$ is *suffix-closed* if there is an assertion A in (future) linear temporal logic such that for every execution e, e satisfies $\mathcal{S}$ if and only if A is true in the terminal configuration of e (for finite executions) or A is infinitely often true in e.

Uniform Agreement: No two processes decide differently.

While in Specification $\mathcal{SP}_2$, we have:

Termination: Every *correct* process eventually decides some value.

Agreement: No two *correct* processes decide differently.

By definition, since all processes are correct, these two specifications are trivially equivalent in a safe environment. By the way, they define the consensus problem. However, if the system is prone to crash failures, an algorithm which satisfied both specifications in a safe environment may satisfy neither of them, while a fault-tolerant algorithm can satisfy Specification $\mathcal{SP}_2$ but not Specification $\mathcal{SP}_1$. Hence those two specifications, although they define the same problem, are not equivalent in a system prone to crash failures.

In order to illustrate our discussion in the context of stabilization, let us focus on the problem of *Mutual Exclusion* (MutEx). In this problem, the code of each process is divided into two sections: the *non-critical* and *critical* sections. Processes alternate between these two sections as follows. Initially, a process executes its *non-critical* section. But, sometimes, the application layer *requests* the process to execute its *critical* section. So, a process should enter its critical section in finite time after each request. The time spent into its critical section is assumed to be a finite, yet unbounded: each process eventually returns to its *non-critical* section code. Moreover, accesses to critical sections should be managed in such way that no two critical sections are executed concurrently by two different processes. The transition from the non-critical section to the critical section is implemented by a special code, called the *entry* section. Similarly, the transition from the critical section to the non-critical section is implemented in another special code, called the *exit* section. The MutEx problem then consists of the design of the *entry* and *exit* sections.

In the self-stabilization literature, the safety part of the MutEx specification is usually defined by using "*static*" condition(s), *i.e.*, by applying a predicate P over the set of system configurations. The subset of configurations such that P is true (resp. false) is usually called the set of *legitimate* (respectively, *illegitimate*) configurations. For instance, the legitimate configurations of a self-stabilizing algorithm are those included in the set of configurations where at most one process is in the critical section. By contrast, the set of illegitimate configurations contains at least the configurations where more than one process are in the critical section. Let us call any specification based on static safety conditions a *static* specification.

Based on this approach, the MutEx problem is usually specified as follows.

Specification 1.1 (Static MutEx)

Safety: *No two processes execute the critical section simultaneously.*

Liveness: *Upon a request, a process enters the critical section in finite time.*

It is straightforward to show the impossibility of designing a snap-stabilizing algorithm satisfying Specification 1.1. Indeed, since several processes may be in their critical section (simultaneously) in the arbitrary initial configuration, the specification can be violated immediately at the beginning of the execution, regardless of the behavior of the algorithm.

Before proposing another specification, let us focus on the notion of a *starting action*. The design of distributed algorithms in a safe environment distinguishes two types of code: the *spontaneous part* where the code is executed following an external (*w.r.t.*, the algorithm) action called *request* (from an operator or another algorithm) and the *message reception part* where the code is executed at the reception of a message. Initialization of any execution of the algorithm is always done by the spontaneous part (if there exist several initiators, they all execute first that part). We call the first action of the spontaneous part the *starting action*. It is clear that any execution in a safe environment always starts with a starting action. Of course, this is generally not the case for stabilizing algorithms, since the first configuration is indeterminate. So, the starting actions may not be the first actions of the execution of a stabilizing algorithm.

We now come back to our MutEx example. Note that the execution of the entry section is triggered by a request (liveness). In other words, every execution of a MutEx algorithm at some process locally follows the same sequential scheme: Request, Entry Section, Critical Section, and Exit Section. So, in this problem, the starting action corresponds to the first action of the entry section. Consider the following new specification where we modify only the safety part:

Specification 1.2 (Dynamic MutEx)

Safety: *If a requesting process p enters the critical section, then p executes the critical section alone.*

Liveness: *Upon a request, a process enters the critical section in finite time.*

By contrast with Specification 1.1, an immediate consequence of Specification 1.2 is that an arbitrary initial configuration where more than one process is executing its critical section is now not illegitimate anymore. Indeed, in such a configuration, no process has actually entered its critical section, *i.e.*, no process has made the transition from the non-critical code to the critical section (using the Entry Section), *i.e.*, no process has executed a starting action. Actually, no request for critical section has even been made. As explained before, in self-/snap- stabilization, the arbitrary initial configuration corresponds to a configuration that can be the result of a finite number of transient faults. So, nothing except transient faults can explain why several processes are executing their critical section in the arbitrary initial configuration.

Safety condition of Specification 1.2 is an implication, where the left part is true if a starting action has been effectively executed before. Therefore, considering Specification 1.2, a configuration cannot be declared *legitimate* or *illegitimate* without considering its past, *i.e.*, the execution prefix that led to that configuration. This is why we call such a specification a *dynamic* specification.

An interesting question arises from the above discussion: "*Does Specification 1.2 define the MutEx safety*?" Our answer is yes. Indeed, in a safe environment, every process which is executing its critical section has previously executed its entry section, and upon a request. So, every algorithm that satisfies Specification 1.1 also satisfies Specification 1.2, and *vice versa*.

Another natural question is the following: "*Can we find a snap-stabilizing solution to the MutEx problem*?" Again, the answer is yes. Two algorithms that are snap-stabilizing for Specification 1.2 are proposed in [12, 26].

To conclude the discussion about the MutEx problem, starting from an arbitrary initial configuration (or equivalently, after the last transient fault ceases), a snap-stabilizing MutEx algorithm does not guarantee that in this configuration, no two processes can be in the critical section. But, it guarantees that upon any (new) request for a process p to enter its critical section, p will enter the critical section within finite time, and p will not do it unless it can do it safely. Specifically, the snap-stabilizing MutEx algorithm will ensure that no other process is in the critical section before allowing p to enter the critical section. A self-stabilizing MutEx algorithm generally does not provide such a safety property, or simply does not consider this issue.

The MutEx example provides a generic approach for writing specifications compatible with snap-stabilization, that can be formulated as follows: *Just recognize the classical starting action in a safe environment, and add it as the condition of the safety*. So, consider the behavior of a distributed algorithm $\mathcal{A}$ beginning from a starting action. If $\mathcal{A}$ is self-stabilizing, then within a finite time (typically, the *stabilization time*), $\mathcal{A}$ will start behaving properly. However, during the stabilization period, behavior of $\mathcal{A}$ is unpredictable, *i.e.*, it may not satisfy its specification. If $\mathcal{A}$ is snap-stabilizing, it starts within finite time and its behavior after the starting action will be as per its specification. This difference shows the extra power of snap-stabilization with respect to self-stabilization: snap-stabilization system provides stronger safety properties.

Contribution We can say from the above discussion that the safety property provided by snap-stabilization is a highly desirable property. Now, given that we define problems in terms of dynamic specification, another interesting question is: "*Can we provide a snap-stabilizing solution to every problem that has a self-stabilizing solution*?"

We provide an answer to this question in the locally shared memory model by proceeding as follows.

We first design a snap-stabilizing *Propagation of Information with Feedback* (PIF) for rooted networks of arbitrary connected topology. We prove this algorithm assuming the distributed unfair daemon, the most general daemon. To the best of our knowledge, this is the only snap-stabilizing PIF algorithm for arbitrary connected rooted networks which has been proven, until now, assuming the distributed unfair daemon. Its time complexity is in $O(N)$ rounds and $O(\Delta \times N^3)$ steps, where N is the number of processes in the system and Δ its degree. Its memory requirement is $O(\log N)$ bits per process. The space and round complexities of our solution asymptotically match those of the previous solutions ([10, 11]). However, contrary to the previous solutions, we could exhibited a bound on its step complexities ($O(\Delta \times N^3)$ steps), as it works under an unfair daemon.

Then, we use this snap-stabilizing PIF to implement snap-stabilizing versions of four fundamental distributed algorithms: *Reset*, *Snapshot*, *Leader Election*, and *Termination Detection*.

Based on these four key algorithms, we design, in the locally shared memory model, a *universal transformer* that provides a snap-stabilizing version of any algorithm which can be self-stabilized using the transformer described

in [28]. Our transformation supports the distributed unfair daemon, the weakest scheduling assumption of the model. This shows that snap-stabilization is as expressive as self-stabilization in the locally shared memory model.

Note that our purpose is only to demonstrate the feasibility of transforming almost any algorithm (specifically, those algorithms that can be self-stabilized) to a corresponding snap-stabilizing algorithm. As a consequence, our method is inefficient due to its versatility. Notice that another efficient, yet non general, transformer has been proposed in the literature [12]. The proposed method allows to build snap-stabilizing algorithms efficient in both space and time, however it only addresses a restricted class of problems, namely mono-initiator wave specifications.

Roadmap In the next section (Section 2), we describe the distributed system and the model we consider in this paper. In Section 3, we formally define the concept of snap-stabilization. In the same section, we also clarify some concepts seemingly similar in the domains of self-stabilization and snap-stabilization. In Section 4, we propose a snap-stabilizing PIF algorithm. Using this algorithm, we then present in Section 5 snap-stabilizing solutions of the Leader Election, Reset, Snapshot, and Termination Detection problems. In Section 6, we show their applications in developing a universal transformer. Finally, we make some concluding remarks in Section 7.

2 Preliminaries

2.1 Distributed Systems

We consider *distributed systems* of N *processes*. Each process p can directly communicate with a subset of other processes, called its *neighbors*. Communication is *bidirectional*. For every process p, we call *degree of* p, noted δ_p, the number of its neighbors. Let $\Delta = \max_{p \in V} \delta_p$ be the degree of the system. The topology of the system is a simple undirected connected graph $G = (V, E)$, where V is the set of processes and E is a set of edges representing (direct) communication between the corresponding adjacent processes. Every process p can distinguish all its neighbors using a local labeling. All labels of p's neighbors are stored into the set $Neig_p$. $Neig_p$ is locally ordered by $\prec_p$. Moreover, we assume that each process p can identify its local label in the set $Neig_q$ of each neighbor q. By an abuse of notation, we use p to designate both the process p itself and its local labels.

2.2 Computational Model

We assume the *locally shared memory model* [3], where each process communicates with its neighbors using a finite set of locally shared registers, henceforth called *variables*. A process can read its own variables and those of its neighbors, but can write only to its own variables. We define a *(distributed) algorithm* to be a collection of N *programs*, each operating on a single process. The program of each process is a finite ordered set of actions, where the ordering defines *priority*. This priority is the order of appearance of actions in the text of the program. A process p is not enabled to execute any action if it is enabled to execute an action of higher priority. Let $\mathcal{A}$ be a distributed algorithm, consisting of a local program $\mathcal{A}(p)$ for each process p. Each action in $\mathcal{A}(p)$ is of the following form:

$$\langle \mathit{label} \rangle \ :: \ \langle guard \rangle \ \rightarrow \ \langle \mathit{statement} \rangle$$

Labels are only used to identify actions. The *guard* of an action in $\mathcal{A}(p)$ is a Boolean expression involving the variables of p and its neighbors. The *statement* of an action in $\mathcal{A}(p)$ updates some variables of p. The *state* of a process in $\mathcal{A}$ is defined by the values of its variables in $\mathcal{A}$. A *configuration* of $\mathcal{A}$ is an instance of the states of processes in $\mathcal{A}$. $\mathcal{C}_\mathcal{A}$ is the set of all possible configurations of $\mathcal{A}$. (When there is no ambiguity, we omit the subscript $\mathcal{A}$.) An action can be executed only if its guard evaluates to $true$; in this case, the action is said to be *enabled*. A process is said to be enabled if at least one of its actions is enabled. We denote by *Enabled*(γ) the subset of processes that are enabled in configuration γ. When the configuration is γ and *Enabled*$(\gamma) \neq \emptyset$, a *daemon* (scheduler) selects a non-empty set $\mathcal{X} \subseteq$ *Enabled*(γ); then every process of $\mathcal{X}$ *atomically* executes its highest priority enabled action, leading to a new configuration γ', and so on. (The daemon realizes the asynchrony of the system.) The transition from γ to γ' is called a *step* (of $\mathcal{A}$). The possible steps induce a binary relation over configurations of $\mathcal{A}$, denoted by $\mapsto$. An *execution* of $\mathcal{A}$ is a maximal sequence of its configurations $e = \gamma_0\gamma_1 \dots \gamma_i \dots$ such that $\gamma_{i-1} \mapsto \gamma_i$ for all $i > 0$. The term "maximal" means that the execution is either infinite, or ends at a *terminal* configuration in which no action of $\mathcal{A}$ is enabled at any process.

Each step from a configuration to another is driven by a daemon. A *daemon* is usually defined in terms of *fairness* and *distribution*. There exist several kinds of fairness assumption. In this paper, we consider the *weak fairness*, and *unfairness* assumptions. Under a *weakly fair* daemon, every continuously enabled process is eventually chosen by the daemon. The *unfair* daemon is the weakest scheduling assumption: it can forever prevent a process from executing an action unless it is the only enabled process. Concerning the *distribution*, we assume that the daemon is *distributed* meaning that, at each step, if one or more processes are enabled, then the daemon chooses at least one (possibly more) of these processes to execute an action.

We say that a process p is *neutralized* in the step $\gamma_i \mapsto \gamma_{i+1}$ if p is enabled in γ_i and not enabled in γ_{i+1}, but does not execute any action between these two configurations. Neutralization of a process can be caused in the following situation: At least one neighbor of p changes its state between γ_i and γ_{i+1}, and this change makes the guards of all actions of p false.

To evaluate time complexity, we use the notion of *round* [32]. This notion captures the execution rate of the slowest process in any execution. Let e be an execution. The first round of an execution e, noted e', is the minimal prefix of e in which every process that is enabled in the initial configuration either executes an action or becomes neutralized. Let e'' be the suffix of e starting from the last configuration of e'. The second round of e is the first round of e'', and so forth.

3 Snap-Stabilization

The goal of this section is twofold. (1) We formally define snap-stabilization (Subsection 3.2). (2) We clarify some concepts seemingly similar in the domains of self-stabilization and snap-stabilization (*e.g.*, *normal* and *legitimate* configurations in Subsection 3.1, and *delay* and *stabilization time* in Subsection 3.3).

First, as in [28], we only deal with problems that can be defined by a *suffix-closed* specification. Second, we consider specifications where *the classical starting action is the condition of the safety* (as presented in Section 1), *i.e.* the safety part of the specification is of the form $P \Rightarrow Q$, where P becomes true when a starting action is executed, and Q checks if some safety requirements are satisfied in the following of the execution.

Consider, for example, a specification $\mathcal{SP}_\mathcal{P}$ of some *finite* problem $\mathcal{P}$. Let $P \Rightarrow Q$ be the safety of $\mathcal{SP}_\mathcal{P}$. Let $\mathcal{A}_{\texttt{self}}$ be an algorithm which is self-, yet not snap-, stabilizing for $\mathcal{SP}_\mathcal{P}$. The first time a starting action of $\mathcal{A}_{\texttt{self}}$ is executed, the initiated computation (which involves all or part of processes) is finite (liveness properties are always satisfied by self-stabilizing algorithms), but may violate Q, and if applicable, $\mathcal{SP}_\mathcal{P}$ is not satisfied. Consequently, to obtain self-stabilization, $\mathcal{A}_{\texttt{self}}$ has to repeat finite computations indefinitely, each computation being initiated by a starting action. Solving this particular drawback is the goal of snap-stabilization. Indeed, snap-stabilization guarantees that after the very first starting action, the (initiated) computation both satisfies the liveness of $\mathcal{SP}_\mathcal{P}$ (in particular, the computation should be finite) and Q. In this case, we say that the (initiated) computation *conforms* to the specification $\mathcal{SP}_\mathcal{P}$.

3.1 Normal vs. Legitimate Configurations

Recall that, as explained in the introduction, the notion of intrinsic legitimate configuration is meaningless in snap-stabilization because it is impossible to verify if the (dynamic) specification is satisfied just by looking at that configuration. (We need to consider the whole execution prefix that led to the configuration.) However, we can characterize a configuration using the notions of *normal* and *abnormal* configurations (defined below), which are specific to the snap-stabilizing paradigm.

In (self- or snap-) stabilizing systems, we consider the system immediately after transient faults cease. That is, we study the system starting from a configuration reached due to the occurrence of transient faults, but from which no fault will ever occur. Due to the effect of the faults, this configuration is arbitrary. Now, this configuration is referred to as an (arbitrary) *initial configuration* of the system because it is the initial point of observation in the proofs.

The notion of "initial configuration" used here should be clearly distinguished from the classical notion of initial configuration used in the non fault-tolerant algorithms. To avoid any confusion, this latter notion will be called *normal initial configuration*. Consider a non fault-tolerant distributed algorithm $\mathcal{A}$ realizing specification $\mathcal{SP}_\mathcal{P}$. As explained in the introduction, the correctness of such an algorithm is established by observing the system from a pre-defined configuration from which the system is supposed to start, the *normal initial configuration*. In this configuration,

initiators are enabled for executing a starting action, and the other processes are quiescent, *i.e.*, they are disabled until being involved in a computation initiated by some initiator.

Of course, such normal initial configurations also exist in (self- and snap-) stabilizing systems. For example, in self-stabilizing token circulation algorithms, the normal initial configuration is usually the configuration where all processes are in the state "idle." Besides, in case of a real deployment, the network should be initialized in the normal initial configuration of the stabilizing algorithm.

In the following, any configuration reachable from a *normal initial configuration* will be called a *normal configuration*. Any execution starting from a normal initial configuration will be termed as a *normal execution*. Considering faulty networks, the system may be in a configuration which is unreachable from a normal initial configuration. We call this type of configurations *abnormal configurations*. Since a configuration is defined by the state of all the processes, we can claim that every process in a normal configuration is a *normal process*, *i.e.*, the state of each process is consistent with those of its neighbors. So, in an abnormal configuration, there exists at least one *abnormal process*, a process whose state is inconsistent with those of its neighbors (nothing but a transient fault can explain this inconsistency).

3.2 Formal Definition of Snap-Stabilization

In [6, 7], Bui *et al* formally defined snap-stabilization as follows:

Definition 3.1 (Snap-stabilization) *A distributed algorithm $\mathcal{A}_{\mathtt{snap}}$ is* snap-stabilizing *for the specification $\mathcal{SP}_{\mathcal{P}}$ if starting from any configuration, every execution of $\mathcal{A}_{\mathtt{snap}}$ satisfies $\mathcal{SP}_{\mathcal{P}}$.*

In the following, we justify why we consider specifications where the classical starting action is the condition of the safety and show the consequence in the proving process.

Consider a specification $\mathcal{SP}_{\mathcal{P}}$ of some problem $\mathcal{P}$, and a non fault-tolerant distributed algorithm $\mathcal{A}$ realizing $\mathcal{SP}_{\mathcal{P}}$. As explained before, the correctness of $\mathcal{A}$ is established by observing the system from a normal initial configuration, from which the system is supposed to start. In this configuration, some processes (called *initiators*) spontaneously start executing the algorithm with a particular portion of code, usually called *initialization*. First, the algorithm designer implicitly assumes that initializations are triggered by some external (with respect to the algorithm) requests, *e.g.*, these requests may be generated by a user or an application. Then, the safety of $\mathcal{SP}_{\mathcal{P}}$ is proven by showing that between the initialization and termination of $\mathcal{A}$, no safety property of $\mathcal{SP}_{\mathcal{P}}$ is ever violated. Now, the system exists before the first initialization, and again the designer implicitly assumes that the safety cannot be violated between the "real" start of the system and the first initialization. This justifies why we consider specifications where the classical starting action is the condition of the safety.

Since we consider specifications where the initialization (handled by the starting actions) is explicitly mentioned, a snap-stabilizing algorithm $\mathcal{A}_{\mathtt{snap}}$ always satisfies its specification if and only if:

1. Starting from any arbitrary configuration, $\mathcal{A}_{\mathtt{snap}}$ can start in finite time using a special action, called *starting action*; such an action is triggered by an external request.

2. Since a starting action is executed, the initiated computation of $\mathcal{A}_{\mathtt{snap}}$ conforms to the specification.

3.3 Delay vs. Stabilization Time

Formally we distinguish three parts of the execution of a distributed self- or snap- stabilizing algorithm as follows. Consider an execution $e_{\mathtt{self}}$ of a self-stabilizing algorithm. First, because the self-stabilization only ensures that the system eventually satisfies the intended specification, $e_{\mathtt{self}}$ should include executions of starting actions infinitely often. Then, it follows from the previous discussion that before $e_{\mathtt{self}}$ starts, the safety of the specification is not violated. Let us denote this part of the execution as a *default prefix*. When $e_{\mathtt{self}}$ starts, the starting action may be executed from an abnormal configuration, and the specification may not be satisfied immediately. However, the specification is eventually true forever: eventually an initiated computation conforms to the specification. So, self-stabilization only guarantees that there exists a suffix of $e_{\mathtt{self}}$ which satisfies the specification. Let us call this suffix a *correct suffix*. The first step of a correct suffix includes a starting action. So, between the end of the default prefix and the correct suffix, the number of times the algorithm initiates (by a starting action) a computation that does not

conform to the specification is finite, yet generally unbounded. Let us call this part of $e_{\texttt{self}}$ the *stabilization factor*. Hence, $e_{\texttt{self}}$ is the concatenation of a default prefix, a stabilization factor, and a correct suffix.

Now, consider an execution $e_{\texttt{snap}}$ of a snap-stabilizing algorithm. By definition, as soon as $e_{\texttt{snap}}$ starts, even if the starting action is executed from an abnormal configuration, the initiated computation conforms to the specification, so the suffix starting from this step is a correct suffix. Thus, $e_{\texttt{snap}}$ is the concatenation of a default prefix and a correct suffix, from where $e_{\texttt{snap}}$ (always) satisfies the specification. Thus, proving an algorithm $\mathcal{A}$ to be snap-stabilizing may involve showing the following two steps. (i) Any execution includes at least one step which contains a starting action (*i.e.*, the default prefix is finite). (ii) Starting from this step, the initiated computation conforms to the specification (*i.e.*, the suffix starting from this step is a correct suffix).

Consider now an external request for a process to initiate an algorithm. We first observe the behavior of a non fault-tolerant distributed algorithm from this point. For instance, assume one requests to print a file, and then makes the same request for a second file. So the printing of the second file is delayed until the previous printing has finished. Take the second request as the initial point of the observation: we observe a delay before the printing algorithm starts for this request.

Observe now the execution of a distributed (self- or snap-) stabilizing algorithm. As above, an initiator may have to wait before starting the algorithm. Precisely, it has to wait until the default prefix is done. By analogy with the previous example, we call *delay* the duration of the default prefix. After the delay, the behavior depends on whether the algorithm is self- or snap-stabilizing. In the case of self-stabilization, the stabilization factor may not be empty. We call the execution time of this factor the *stabilization time*.[2] In snap-stabilization, the stabilization factor is empty, and consequently, the self-stabilization time is null.

We should emphasize here that the notion of delay and stabilization time are clearly different. The impact of the delay is to slow down the algorithm, whereas during the stabilization time, the specification of the algorithm is violated.

4 Snap-Stabilizing PIF

The concept of *Propagation of Information with Feedback* (PIF), also called *Wave Propagation*, has been introduced by Chang [33] and Segall [34]. PIF has been extensively studied in the distributed literature because many fundamental algorithms, *e.g.*, *Reset*, *Snapshot*, *Leader Election*, *Termination Detection*, etc. can be solved using a PIF-based approach. Two snap-stabilizing versions of the PIF were presented for arbitrary networks [10, 11]. The major advantage of the snap-stabilizing solution proposed here is that it is proven under the distributed unfair daemon. It is also important to note that we obtain this result without degrading the performance. The round and space complexities of our solution match the previous results.

The PIF scheme can be informally described as follows. A process, called *initiator*, starts the first phase of the PIF wave by broadcasting a message *m* in the network (this phase is called *broadcast phase*). Then, each non-initiator acknowledges the receipt of *m* to the initiator (this phase is called *feedback phase*). The PIF wave terminates when the initiator has received an acknowledgment from all other processes. In arbitrary distributed systems, any process may need to initiate a PIF wave. Thus, any process can be the initiator of a PIF wave and several PIF algorithms may run simultaneously. To cope with the concurrent executions, every process maintains the identity of the initiators.

In this section, we consider the problem in a general setting of the PIF scheme where we assume that the PIF is initiated by a process called the *root*, denoted by *r*. We can formally specify a PIF wave as follows:

Specification 4.1 (PIF Wave) *A finite execution $e = \gamma_0, \ldots, \gamma_i, \gamma_{i+1}, \ldots, \gamma_t$ is called a* PIF Wave *if and only if the following condition is true:*
If r broadcasts a message m *in the computation step $\gamma_0 \mapsto \gamma_1$, then:*

[PIF1] *For each $p \neq r$, there exists a unique $i \in [1, t-1]$ such that p receives* m *in $\gamma_i \mapsto \gamma_{i+1}$, and*

[PIF2] *In γ_t, r receives an acknowledgment of the receipt of* m *from every process $p \neq r$.*

Remark 4.1 *In practice, to prove that a PIF algorithm is snap-stabilizing, we have to show that every execution of the algorithm satisfies the following two conditions: (i) if r has a message* m *to broadcast, it will do so in a finite time, and (ii) starting from any configuration where r broadcasts* m*, the system satisfies Specification 4.1.*

[2]Notice that, in the self-stabilization literature, the delay is included in the stabilization time. We conjecture that the stabilization time complexity is of the same order of magnitude with or without including the delay. Actually, we did not find any counter-example until now.

Algorithm 4.1 $\mathcal{PIF}$ for $p = r$

Input: $Neig_p$: set of (locally) ordered neighbors of p;
Constants: $Par_p = \perp$; $L_p = 0$;
Variables: $S_p \in \{B, F, P, C\}$; $Que_p \in \{Q, R, A\}$;

Set Macro:
$Children_p \quad = \quad \{q \in Neig_p :: (S_q \neq C) \wedge (Par_q = p) \wedge (L_q = L_p + 1) \wedge [(S_q \neq S_p) \Rightarrow (S_p \in \{B, P\} \wedge S_q = F)]\}$;

General Predicates:
$CFree(p) \equiv (\forall q \in Neig_p :: S_q \neq C)$
$Leaf(p) \equiv [\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (Par_q \neq p)]$
$BLeaf(p) \equiv (S_p = B) \wedge [\forall q \in Neig_p :: (Par_q = p) \Rightarrow (S_q = F)]$
$AnswerOk(p) \equiv (Que_p = A) \wedge [\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (Que_q = A)]$
Guards:
$Broadcast(p) \equiv (S_p = C) \wedge Leaf(p)$
$Feedback(p) \equiv BLeaf(p) \wedge CFree(p) \wedge AnswerOk(p)$
$Preclean(p) \equiv (S_p = F) \wedge [\forall q \in Neig_p :: (Par_q = p) \Rightarrow (S_q \in \{F, C\})]$
$Cleaning(p) \equiv (S_p = P) \wedge Leaf(p)$
$Reset(p) \equiv (S_p \in \{B, F\}) \wedge [(S_p = B) \Rightarrow CFree(p)] \wedge [[(Que_p = Q) \wedge (\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (Que_q \in \{Q, R\}))]$
$\vee [(Que_p = A) \wedge (\exists q \in Neig_p :: (S_q \neq C) \wedge ((Que_q = Q) \vee (q \in Children_p \wedge Que_q = R)))]]$
$Answer(p) \equiv (S_p \in \{B, F\}) \wedge [(S_p = B) \Rightarrow CFree(p)] \wedge (Que_p = R) \wedge (\forall q \in Children_p :: Que_q \in \{W, A\})$
$\wedge [\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (Que_q \neq Q)]$

Actions:
PIF Part:
$B\text{-}action :: Broadcast(p) \to S_p := B; Que_p := Q;$ /* *Starting Action* */
$F\text{-}action :: Feedback(p) \to S_p := F;$
$P\text{-}action :: Preclean(p) \to S_p := P;$
$C\text{-}action :: Cleaning(p) \to S_p := C;$
Question Part:
$QR\text{-}action :: Reset(p) \to Que_p := R;$
$QA\text{-}action :: Answer(p) \to Que_p := A;$

4.1 The Algorithm

The algorithm provided in Algorithms 4.1 and 4.2 implements a snap-stabilizing PIF. This algorithm, referred to as $\mathcal{PIF}$ in the following, is divided in three parts: the *PIF*, *Question*, and *Correction Part*.

The *PIF Part* is the main part of the algorithm. It implements the actions corresponding to each of the three phases of the PIF wave, *i.e.*, the *broadcast* phase, the *feedback* phase following the broadcast, and the *cleaning phase* which cleans the trace of the feedback so that the root becomes able to broadcast another message, if necessary.

A process initiates the feedback phase when it cannot broadcast the message because all its neighbors have already received the message from some other neighbors. However, as we consider here that the initial configuration can be arbitrary, the system can initially contain some processes that broadcast a message whose source is not the root. Thus, when a process is ready to initiate the feedback phase because all its neighbors seem to have received the message (from the root), it must be sure that its neighborhood have really received the message of the root and, so, none of its neighbors is participating to the broadcast of any erroneous message. That is the goal of the *Question Part*. A question is emitted to the root by a process p each time it receives a new broadcast message. Then, p waits an authorization from the root to execute its feedback phase: this authorization actually corresponds to the reception by p and its neighbors of a positive answer from the root.

Finally, note that the *Correction Part* contains the actions dealing with the error correction.

We now describe in details the three parts of Algorithm $\mathcal{PIF}$.

4.1.1 PIF Part

Let us consider as normal initial configuration any configuration γ where every process p satisfies $S_p = C$. The variable S_p gives the status of p *w.r.t.* the PIF and status C (which stands for clean) means that p is not involved into and PIF wave. In γ, $B\text{-}action$ at *r* is the only enabled action (B means broadcast). So, *r* executes $B\text{-}action$ in the first step: *r* switches to the broadcast phase by executing $S_r := B$ (when $S_r = B$, *r* is supposed to broadcast a message to all its neighbors) and initiates a *question* by $Que_r := Q$. When a process p waiting for a message ($S_p = C$) has one of its neighbors q involved in the broadcast phase ($S_q = B$), p receives the message from q by executing $B\text{-}action$: p switches to the broadcast phase ($S_p := B$), initiates a question ($Que_p := Q$), points out to q using Par_p, and sets its level L_p to $L_q + 1$. (Typically, L_p contains the length of the path followed by the broadcast message from *r* to p.) Process p is now in the broadcast phase and is supposed to broadcast the message to all its neighbors except

Par_p. Using this mechanism, a spanning tree rooted at r (*w.r.t.* the Par variables) is dynamically built during the broadcast phase. Let us call $\texttt{Tree}(r)$ this tree. Eventually, the broadcast reaches some process p that cannot broadcast the message because all its neighbors have already received the message from some other neighbors ($\forall q \in Neig_p$, $S_q \neq C \wedge Par_q \neq p$). Such a process p is called a *leaf* and waits for an authorization from the root to switch to the feedback phase. This authorization is a positive answer to p and its neighbors to the question previously asked by p ($AnswerOk(p)$). After receiving an authorization, p switches to the feedback phase by F-$action$ ($S_p := F$). The feedback phase is then propagated up into $\texttt{Tree}(r)$ as follows: A non-leaf process q switches to the feedback phase when the following conditions are true:

1. All its children in $\texttt{Tree}(r)$ satisfy $S = F$ ($BLeaf(q)$) meaning that all the processes it has sent the message have executed their feedback.
2. None of its neighbors satisfies $S = C$ ($CFree(q)$), *i.e.*, none of its neighbors is waiting for a message.
3. It is authorized by the root ($AnswerOk(q)$) to ensure that all its neighbors involved in a PIF wave effectively participates in the PIF wave from r.

Thanks to that, all processes eventually participate in both broadcast and feedback phases, and the feedback phase eventually ends at r. At that point, the cleaning phase needs to be executed so that the root can broadcast another message. The cleaning phase just consists of a PIF on $\texttt{Tree}(r)$, the tree built during the broadcast phase. Such a PIF is initiated by r when it detects the end of the feedback phase: r broadcasts a P (meaning $Preclean$) value in $\texttt{Tree}(r)$ towards the leaves and following the Par pointer (see P-$action$). The corresponding feedback phase then cleans the tree in a bottom-up fashion (C-$action$).

4.1.2 Question Part

The questions (and the corresponding answers) are used to prevent the following problem. When the system starts from an arbitrary configuration, a process p may receive a message from r while one of its neighbors q satisfies $S_q \in \{B, F\} \wedge q \notin \texttt{Tree}(r)$. Actually, q is in a tree rooted at a process different from r, called an *abnormal tree*. We will see later that such abnormal trees are eventually deleted using the *Correction Part*. But, while q is in an abnormal tree, p must not switch to the feedback phase. Otherwise, q will not receive the broadcast message from p, and as a consequence, q may never receive the message sent by r. Hence, the goal of the question (and its answer) is to ensure that p switches to the feedback phase only when all its neighbors are in $\texttt{Tree}(r)$ (*i.e.*, only after all its neighbor received the message from r). The question mechanism is implemented using the Que variables. $Que_p \in \{Q, R, A\}$ for $p = r$ and $Que_p \in \{Q, R, W, A\}$ for $p \neq r$. The Q and R values are used to reset the part of the network relevant to a *question*. The W value corresponds to the request of a process: "Do you authorize me to feedback?". The A value corresponds to the positive answer sent by r: r is the only process able to initiate an A value.

We now explain how this phase works. A *question* is initiated at p by executing $Que_p := Q$ each time p switches to the broadcast phase. This action forces all its neighbor q satisfying $S_q \neq C$ to reset Que_q to R (QR-$action$). After every q (every neighbor of p) has reset, p also executes QR-$action$. The R values are then propagated up the trees of p and every q (and only the trees). By this mechanism, all A values (in particular, the A values present since the initial configuration) possibly in the path from p (resp. q) to its source (*w.r.t.* the Par variable) are erased . So, from that point onwards, when an A value reaches a requesting process or one of its neighbors, this value must have been sent by r and the process obviously is in $\texttt{Tree}(r)$.

As we have discussed before, eventually the broadcast phase reaches some leaves of $\texttt{Tree}(r)$ and these leaves need an authorization from the root to start the feedback phase. In this case, each leaf p executes $Que_p := W$ (QW-$action$), providing that $Que_p = Que_{Par_p} = R$ and meaning that p is now waiting for an answer from r. The W value is then propagated up in the tree of p (and only this tree) as follows. A non-leaf process q can execute QW-$action$ if $Que_p = Que_{Par_p} = R$, all its children satisfy $Que \in \{W, A\}$, and no neighbor has $S = C$. When the W values reaches all the children of r, $Que_r = R$ and r executes QA-$action$. r broadcasts an answer A in its tree, and so on. Hence, every time a process p initiates a question, we are sure that p and its neighbors that were in $\texttt{Tree}(r)$ when the question was sent eventually receive an A value. On the contrary, the neighbors of p that were in abnormal trees do not receive any A value before leaving their trees (using the *Correction Part*). Suppose now that a neighbor q of p such that $S_q = C$ attaches to a tree before the question was answered. Then q executes $Que_q := Q$, and either q is in $\texttt{Tree}(r)$ and q eventually receives an A value, or q is in an abnormal tree and Que_q remains different from A until q

Algorithm 4.2 $\mathcal{PIF}$ for $p \neq r$

Input: $Neig_p$: set of (locally) ordered neighbors of p;
Variables: $S_p \in \{B, F, P, C, EB, EF\}$; $Par_p \in Neig_p$; $L_p \in [1 \ldots \ell]$ with $\ell \geq N$; $Que_p \in \{Q, R, W, A\}$;

Set Macros:

$Children_p = \{q \in Neig_p :: (S_q \neq C) \wedge (Par_q = p) \wedge (L_q = L_p + 1) \wedge [(S_q \neq S_p) \Rightarrow ((S_p \in \{B, P\} \wedge S_q = F) \vee (S_p = EB))]\}$;
$Pre_Potential_p = \{q \in Neig_p :: S_q = B \wedge L_q < \ell\}$;
$Potential_p = \{q \in Pre_Potential_p :: \forall q' \in Pre_Potential_p, L_q \leq L_{q'}\}$;

General Predicates:

$CFree(p) \equiv (\forall q \in Neig_p :: S_q \neq C)$
$Leaf(p) \equiv [\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (Par_q \neq p)]$
$BLeaf(p) \equiv (S_p = B) \wedge [\forall q \in Neig_p :: (Par_q = p) \Rightarrow (S_q = F)]$
$AnswerOk(p) \equiv (Que_p = A) \wedge [\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (Que_q = A)]$
$GoodS(p) \equiv (S_p = C) \vee [(S_{Par_p} \neq S_p) \Rightarrow ((S_{Par_p} = EB) \vee (S_p = F \wedge S_{Par_p} \in \{B, P\}))]$
$GoodL(p) \equiv (S_p \neq C) \Rightarrow (L_p = L_{Par_p} + 1)$
$AbRoot(p) \equiv \neg GoodS(p) \vee \neg GoodL(p)$

Guards:

$EFAbRoot(p) \equiv (S_p = EF) \wedge AbRoot(p) \wedge [\forall q \in Neig_p :: (Par_q = p \wedge L_q > L_p) \Rightarrow (S_q \in \{EF, C\})]$
$EBroadcast(p) \equiv (S_p \in \{B, F, P\}) \wedge [\neg AbRoot(p) \Rightarrow (S_{Par_p} = EB)]$
$EFeedback(p) \equiv (S_p = EB) \wedge [\forall q \in Neig_p :: (Par_q = p \wedge L_q > L_p) \Rightarrow (S_q \in \{EF, C\})]$
$Broadcast(p) \equiv (S_p = C) \wedge (Potential_p \neq \emptyset) \wedge Leaf(p)$
$Feedback(p) \equiv BLeaf(p) \wedge CFree(p) \wedge AnswerOk(p)$
$Preclean(p) \equiv (S_p = F) \wedge (S_{Par_p} = P) \wedge [\forall q \in Neig_p :: (Par_q = p) \Rightarrow (S_q \in \{F, C\})]$
$Cleaning(p) \equiv (S_p = P) \wedge Leaf(p)$
$Reset(p) \equiv (S_p \in \{B, F\}) \wedge [(S_p = B) \Rightarrow CFree(p)] \wedge [[(Que_p = Q) \wedge (\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (Que_q \in \{Q, R\}))]$
$\quad \vee [(Que_p \in \{W, A\}) \wedge (\exists q \in Neig_p :: (S_q \neq C) \wedge ((Que_q = Q) \vee (q \in Children_p \wedge Que_q = R)))]]$
$Wait(p) \equiv (S_p \in \{B, F\}) \wedge [(S_p = B) \Rightarrow CFree(p)] \wedge (Que_p = R) \wedge (Que_{Par_p} = R)$
$\quad \wedge (\forall q \in Children_p :: Que_q \in \{W, A\}) \wedge (\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (Que_q \neq Q))$
$Answer(p) \equiv (S_p \in \{B, F\}) \wedge [(S_p = B) \Rightarrow CFree(p)] \wedge (Que_p = W) \wedge (Que_{Par_p} = A)$
$\quad \wedge (\forall q \in Children_p :: Que_q \in \{W, A\}) \wedge (\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (Que_q \neq Q))$

Actions:

Correction Part:

$EC\text{-}action :: EFAbRoot(p) \rightarrow S_p := C$;
$EB\text{-}action :: EBroadcast(p) \rightarrow S_p := EB$;
$EF\text{-}action :: EFeedback(p) \rightarrow S_p := EF$;

PIF Part:

$B\text{-}action :: Broadcast(p) \rightarrow S_p := B; Par_p := \min_{\prec_p}(Potential_p); L_p := L_{Par_p} + 1; Que_p := Q$;
$F\text{-}action :: Feedback(p) \rightarrow S_p := F$;
$P\text{-}action :: Preclean(p) \rightarrow S_p := P$;
$C\text{-}action :: Cleaning(p) \rightarrow S_p := C$;

Question Part:

$QR\text{-}action :: Reset(p) \rightarrow Que_p := R$;
$QW\text{-}action :: Wait(p) \rightarrow Que_p := W$;
$QA\text{-}action :: Answer(p) \rightarrow Que_p := A$;

leaves its tree. Hence, when a process switches to the broadcast phase from the root, it does not feedback before all its neighbor are in $\texttt{Tree}(r)$.

4.1.3 Correction Part

The error correction code deals with the *abnormal processes* p such that $S_p \neq C$ and $p \notin \texttt{Tree}(r)$. The abnormal processes are arranged into abnormal trees rooted at some non-root processes satisfying $AbRoot$, the *abnormal roots*. An abnormal root is a non-root process in an incoherent state *w.r.t.* its parent. A process cannot reach this state in a normal execution. For example, any process p that does not satisfy $AbRoot(p)$ verifies the following conditions:

1. If p is in the broadcast phase, then its parent is in the broadcast phase too.
2. If p is in the feedback phase, then its parent is in the broadcast phase, the feedback phase, or is involved in the $Preclean$ process (the broadcast of the cleaning phase).
3. If p satisfies $S_p = P$ ($Preclean$), then its parent is also involved in the $Preclean$ process.
4. If p is involved in the PIF wave ($S_p \neq C$), then its level L_p must be equal to one plus the level of its parent.

The other conditions that p has to verify if it is not an abnormal root are related to the correction process and will be presented later.

Before explaining the correction mechanism, let us now clarify the notion of *tree*. We consider here two kinds of trees: the *normal* and *abnormal trees*. The normal tree is the only tree rooted at *r*. An abnormal tree is a tree rooted at

a non-root process satisfying $AbRoot$. Let p be a process such that $(p = r) \vee AbRoot(p)$. $\forall q \in V$, $q \in \texttt{Tree}(p)$ if and only if there exists a sequence of processes $(p_0 = p)$, ..., p_i, ..., $(p_k = q)$ such that, $\forall i \in [1...k]$, $p_i \in Children_{p_{i-1}}$ (among the neighbors designating p_{i-1} with Par only those satisfying $S \neq C \wedge \neg AbRoot$ are considered as p_{i-1} children).

The error correction consists of the removal of all the abnormal trees. To remove an abnormal tree $\texttt{Tree}(p)$, we cannot simply set S_p to C. Since $\texttt{Tree}(p)$ may contain several processes, if we simply set S_p to C, p can participate again in the broadcast of the tree it was the root of. This scheme can be repeated many times, and may slow down the progression of the normal tree $\texttt{Tree}(r)$. We solve this problem by executing a paralyzing PIF on the abnormal trees before removing them. To apply this method, we use two additional states in the S variables: EB and EF, for $p \neq r$ only (EB and EF respectively means error broadcast and error feedback). If p is an abnormal root, it broadcasts the value EB in the S variable of its tree (EB-$action$). When p receives feedback (EF-$action$), p knows that all the processes q of its tree satisfy $S_q = EF$ and no process can now receive the broadcast phase from any q (indeed, $S_q \neq B$, $\forall q$). Process p can then leave its tree (each paralyzed tree is removed in a top-down fashion using EC-$action$), and will try to receive the broadcast from one of the processes q only when q will participate in another broadcast. In this manner, all abnormal trees eventually disappear from the system.

Finally, due to the error correction, any process p must satisfy some other conditions so that it is not considered an abnormal root.

5. If p is in the broadcast phase of the error correction ($S_p = EB$), then its parent is also in the broadcast phase of the error correction.

6. If p is in the feedback phase of the error correction process ($S_p = EF$), then its parent is either in the broadcast or in the feedback phase of the error correction process.

4.2 Proof of Snap-Stabilization of Algorithm $\mathcal{PIF}$

We prove that Algorithm $\mathcal{PIF}$ implements a snap-stabilizing PIF assuming a distributed unfair daemon in two steps.

(i) We first prove that Algorithm $\mathcal{PIF}$ is snap-stabilizing for the PIF specification — Specification 4.1, page 10 — under a (distributed) weakly fair daemon (note that this daemon is stronger than the unfair daemon).

(ii) We then show that each wave executed by Algorithm $\mathcal{PIF}$ contains a finite number of steps.

By these two claims, it is clear that Algorithm $\mathcal{PIF}$ is a snap-stabilizing PIF for a distributed unfair daemon. Indeed, by (ii) we can claim that an unfair daemon cannot prevent any PIF wave from being executed forever, and by (i) we can claim that Algorithm $\mathcal{PIF}$ satisfies the PIF specification beginning from the first wave.

4.2.1 Definitions

Below we define some items used in proofs and show some of their characteristics.

Definition 4.1 (Path) *The sequence of processes* $\texttt{P} = p_0, p_1, p_2, \ldots, p_k$ *is called a* path *if* $\forall i$, $1 \leq i \leq k$, $p_i \in Neig_{p_{i-1}}$. *The processes* p_0 *and* p_k *are respectively termed as the* initial *and* final extremities *of* $\texttt{P}$. *We denote by* $|\texttt{P}|$ *the length of* $\texttt{P}$ *(*$= k$*).*

In the following definition, we define a special path by using the Par variables linking a process to the root of the (normal or abnormal) tree it belongs.

Definition 4.2 (PPath) *For every process* p *such that* $S_p \neq C$, $\texttt{PPath}(p)$ *is the unique path* $p_0, p_1, p_2, \ldots, (p_k = p)$ *satisfying the following conditions:*

1. $\forall i$, $1 \leq i \leq k$, $(S_{p_i} \neq C) \wedge (Par_{p_i} = p_{i-1}) \wedge \neg AbRoot(p_i)$.

2. $(p_0 = r) \vee AbRoot(p_0)$.

Using the notion of $\texttt{PPath}$ we now formally define the notion of *tree*.

Definition 4.3 (Tree) *For every process p such that ($p = r$) $\vee$ $AbRoot(p)$, we define* $\texttt{Tree}(p)$ *as the set of processes such that:* $\forall q \in V$, $q \in \texttt{Tree}(p)$ *if and only if* $S_q \neq C$ *and p is the initial extremity of* $\texttt{PPath}(q)$.

Notation 4.1 *Let T be a tree. Let $p \in T$ and $p_0, p_1, p_2, \ldots$ ($p_k = p$) its* PPath. $\forall i$, $1 \leq i \leq k$, *p_{i-1} is the* parent *of p_i in T. Conversely, p_i is a* child *of p_{i-1} in T. The* height *of p_i in T ($= |\texttt{PPath}(p_i)|$) is denoted by $h(p_i)$. The* height *of T, noted H, is equal to* $\texttt{max}(\{h(q), q \in T\})$.

We now distinguish two kinds of trees: the *normal* and *abnormal* trees.

Definition 4.4 (Normal Tree) *A tree containing only processes p such that ($p = r$) $\vee$ $\neg AbRoot(p)$ is called a* normal tree.

Observation 4.1 *By definition, the system always contains one normal tree: the tree rooted at r. In the case where $S_r = C$,* $\texttt{Tree}(r)$ *is still defined but* $\texttt{Tree}(r) = \emptyset$.

Definition 4.5 (Abnormal Tree) *Any tree rooted at a process other than r is called an* abnormal tree.

In the following two definitions, we introduce the notions of `Alive` and `Dead` trees. These two notions allows us to distinguish a tree that can still grow (`Alive`) from a tree that cannot (`Dead`).

Definition 4.6 (Alive) *A tree T satisfies* $\texttt{Alive}(T)$ *(or is called* `Alive`*) if and only if* $\exists p \in T$ *such that* $S_p = B$.

Definition 4.7 (Dead) *A tree T satisfies* $\texttt{Dead}(T)$ *(or is called* `Dead`*) if and only if* $\neg\texttt{Alive}(T)$.

Observation 4.2 *No process can attach to a* `Dead` *tree.*

The following definition characterizes the paralyzing processes of abnormal trees.

Definition 4.8 (E-Dead) *A tree T satisfies* $\texttt{E-Dead}(T)$ *(or is called* E-dead*) if and only if* $\forall p \in T$, $S_p \in \{EB, EF\}$.

Observation 4.3 $\texttt{E-Dead}(T) \Rightarrow \texttt{Dead}(T)$.

Definition 4.9 (S-Trace) *Let Y be a tuple of processes ($Y = (p_0, p_1, \ldots, p_k)$).* $\texttt{S-Trace}(Y) = S_0 S_1 \ldots S_k$ *is the sequence of the values of Variable S of processes p_i ($i = 0 \ldots k$).*

The fact that any process at height $h > 0$ in a tree satisfies $\neg AbRoot$ leads to the following two observations. These observations can be easily verified by induction on the height of the processes starting from height one.

Observation 4.4 *The normal tree,* $\texttt{Tree}(r)$*, always satisfies one of the two following cases:*

1. $\texttt{Tree}(r) = \emptyset \wedge S_r = C$, *or*
2. $\forall p \in \texttt{Tree}(r)$, $\texttt{S-Trace}(\texttt{PPath}(p)) \in B^* F^* \cup P^* F^*$.

Observation 4.5 *For an abnormal tree T,* $\forall p \in T$, $\texttt{S-Trace}(\texttt{PPath}(p)) \in EB^* EF^* \cup EB^* B^* F^* \cup EB^* P^* F^*$.

4.2.2 Proof assuming a Weakly Fair Daemon

We first prove that any execution of Algorithm $\mathcal{PIF}$ is deadlock-free. To prove that, we consider two cases. We first analyze the configurations containing some abnormal trees (Lemma 4.1), then the configurations containing no abnormal tree (Lemma 4.2).

Lemma 4.1 *In any configuration containing some abnormal trees, there exists at least one enabled process.*

Proof. Assume by contradiction, that there exists a configuration γ containing some abnormal trees where no process is enabled. Among the abnormal trees in γ, consider $\texttt{Tree}(ar)$ with the root ar with the maximal level value L_{ar}.

If there exists a process p in $\texttt{Tree}(ar)$ such that $S_p \in \{B,F,P\}$, then there exists $q \in \texttt{PPath}(p)$ such that $S_q \in \{B,F,P\} \wedge (q = ar \vee S_{Par_q} = EB)$ by Observation 4.5. In both cases, EB-*action* is enabled at q, a contradiction.

So, by the hypothesis of the contradiction, every process p in $\texttt{Tree}(ar)$ satisfies $S_p \in \{EB,EF\}$: $\texttt{Tree}(ar)$ is E-Dead and $\texttt{S-Trace}(\texttt{PPath}(p)) = EB^*EF^*$ for any p by Observation 4.5. If $S_{ar} = EF$, then every process p in $\texttt{Tree}(ar)$ satisfies $S_p = EF$. Moreover, every neighbor q of ar, such that $(Par_q = ar) \wedge (L_q > L_{ar})$ is in $\texttt{Tree}(ar)$ if $S_q \neq C$. Otherwise, q satisfies $AbRoot(q)$, and as ar is the abnormal root with the maximal level value, we obtain a contradiction. Hence, every neighbor q of ar such that $(Par_q = ar) \wedge (L_q > L_{ar})$ satisfies $S_q \in \{C,EF\}$ and EC-*action* is enabled at ar, a contradiction. So, by the hypothesis of the contradiction again, we have $S_{ar} = EB$ and we can deduce that there exists a process p in $\texttt{Tree}(ar)$ such that $(S_p = EB) \wedge (\forall q \in Children_p, S_q = EF)$. Similarly, we can conclude that EF-*action* is enabled at p, a contradiction.

Hence, in any configuration containing some abnormal trees, there exists at least one enabled process, a contradiction. □

Lemma 4.2 *In any configuration containing no abnormal trees, there exists at least one enabled process.*

Proof. Assume, by contradiction, that there exists a configuration γ containing no abnormal trees where no process is enabled. In this case, $\texttt{Tree}(r)$ is the only tree in the system.

If $S_r = C$ (the normal tree is empty), then every process p satisfies $S_p = C$ and B-*action* is enabled at r, a contradiction.

So, $S_r \neq C$ and every process p in $\texttt{Tree}(r)$ satisfies $\texttt{S-Trace}(\texttt{PPath}(p)) \in B^*F^* \cup P^*F^*$ by Observation 4.4.

- *Assume that* $\texttt{Tree}(r)$ *is* Dead. In this case, $\texttt{S-Trace}(\texttt{PPath}(p)) = P^*F^*$ holds for every process p in $\texttt{Tree}(r)$. If there exists a process p in $\texttt{Tree}(r)$ such that $S_p = F$, then there exists a process q in $\texttt{PPath}(p)$ having its P-*action* enabled. Otherwise, $(\forall p \in \texttt{Tree}(r), S_p = P)$, C-*action* is enabled at each leaf of $\texttt{Tree}(r)$. Hence, if $\texttt{Tree}(r)$ is Dead, there exists at least one enabled process, a contradiction.

- *Assume that* $\texttt{Tree}(r)$ *is* Alive. Then there exists a process p in $\texttt{Tree}(r)$ such that $S_p = B$ by Definition 4.6 and every process q in $\texttt{Tree}(r)$ satisfies $S_q \in \{B,F\}$ by Observation 4.4. Also, note that every process such that $S_p = B$ also satisfies $CFree(p)$. Otherwise, there exists at least one neighbor of p with its B-*action* enabled.

  In order to obtain the contradiction, we now show that if there exists a process p such that $(S_p \neq C) \wedge (Que_p \neq A)$, then there exists at least one enabled action of the question part which is enabled. To prove that, we focus on the Que variables.

  - Assume that there exists a process p such that $(S_p \neq C) \wedge (Que_p = Q)$. If $\exists q \in Neig_p$ such that $(S_q \neq C) \wedge (Que_q \notin \{Q,R\})$, then QR-*action* is enabled at q, a contradiction. Otherwise, $(\forall q \in Neig_p, (S_q \neq C) \Rightarrow (Que_q \in \{Q,R\}))$, QR-*action* is enabled at p, a contradiction. Hence, every process p satisfies $(S_p \neq C) \Rightarrow (Que_p = Q)$.
  - Assume that there is a process p such that $(S_p \neq C) \wedge (Que_p = R)$. Suppose $p \neq r$. If $Que_{Par_p} \neq R$, then QR-*action* is enabled at Par_p. So, $\forall q \in \texttt{PPath}(p)$, $Que_q = R$. In this case, at least one of those processes p satisfies $\forall q \in Children_p, Que_q \in \{W,A\}$. Moreover, we already know that if $S_p = B$, p satisfies $CFree(p)$. So, QW-*action* is enabled at p, a contradiction. Hence, for every process $p \neq r$, we have $(S_p \neq C) \Rightarrow (Que_p \neq R)$. Let us now assume that $(S_r \neq C) \wedge (Que_r = R)$. As $\forall q \in Children_r$, $Que_r \in \{W,A\}$ and $(S_r = B) \Rightarrow CFree(r)$, QA-*action* is enabled at r, a contradiction.
  - Finally, assume that there exists a process p such that $(S_p \neq C) \wedge (Que_p = W)$. From the previous cases, we know that for every process p, $(S_p \neq C) \Rightarrow (Que_p \in \{W,A\})$, more specifically, $(S_r \neq C) \Rightarrow (Que_r = A)$. Now, similar to the previous case, it is easy to see that if there exists some processes such that $(S \neq C) \wedge (Que = W)$, at least one has its QW-*action* enabled, a contradiction.

  Hence, every process p such that $S_p \neq C$ satisfies $Que_p = A$, and so, p also satisfies $AnswerOk(p)$. We assumed that there exists at least one process p in $\texttt{Tree}(r)$ such that $S_p = B$. Among the processes p satisfying $S_p = B$, at least one p' satisfies $BLeaf(p')$ (by Observation 4.4, every process p in $\texttt{Tree}(r)$ satisfies $\texttt{S-Trace}(\texttt{PPath}(p)) = B^*F^*$). Because p' also satisfies $CFree(p')$, F-*action* is enabled at p', a contradiction.

Hence, in any configuration containing no abnormal trees, there exists at least one enabled process, a contradiction. □

By Lemmas 4.1 and 4.2, we can claim the following result:

Theorem 4.1 *In any configuration, there exists at least one enabled process.*

Lemmas 4.3 to 4.7 are used to show that the network contains no abnormal tree in $O(N)$ rounds (Lemma 4.7).

Lemma 4.3 *If EB-action is enabled at p, it remains enabled until it is executed by p.*

Proof. Assume by contradiction that EB-*action* is enabled at p in γ and not in the next configuration γ', but p did not execute EB-*action* in the step $\gamma \mapsto \gamma'$. Let $q \in Neig_p$ such that $Par_p = q$ ($p \neq r$). $EBroadcast(p)$ involves variables S and L of p and q only. Also, EB-*action* is the enabled action at p with the highest priority (EC-*action* and EB-*action* cannot be enabled at p at the same time). So, if p did not move in $\gamma \mapsto \gamma'$, p still satisfies $S_p \in \{B,F,P\}$, $Par_p = q$ in γ', and q executes an action in $\gamma \mapsto \gamma'$ which updates the value of S_q and/or L_q, so that $\neg EBroadcast(p)$ in γ'. Finally, from $EBroadcast(p)$, we know that $S_p \in \{B,F,P\} \wedge [AbRoot(p) \vee (\neg AbRoot(p) \wedge S_{Par_p} = EB)]$ in γ. Let us now study the following two cases:

- *$AbRoot(p)$ in γ.* $(AbRoot(p) \wedge S_p \in \{B,F,P\}) \Rightarrow EBroadcast(p)$, so $\neg AbRoot(p)$ in γ'. Assume that $\neg GoodL(p)$ in γ (note that $AbRoot(p) \equiv \neg GoodS(p) \vee \neg GoodL(p)$). As $S_p \in \{B,F,P\}$, $Par_p = q$ and $GoodL(p)$ in γ', $L_p = L_q + 1$ in γ'. q must execute the B-*action*. Now, as $S_p \in \{B,F,P\}$ and $Par_p = q$ in γ, q satisfies $\neg Leaf(q)$ and B-*action* is disabled at q in γ, a contradiction. Hence, assume that $GoodL(p) \wedge \neg GoodS(p)$ in γ and $GoodS(p)$ in γ'. In this case, $\texttt{S-Trace}(q,p) \in \{BP,FB,CB,CF,CP,PB,FP,EFB,EFF,EFP\}$ in γ and EB-*action* is the only action that q may execute (in particular, B-*action* is disabled at q because $\neg Leaf(q)$). Now, if q executes EB-*action*, then $S_q = EB$ in γ' and, as $S_p \in \{B,F,P\}$ in γ' (p did not move in $\gamma \mapsto \gamma'$), EB-*action* is still enabled at p in γ', a contradiction.
- *$(\neg AbRoot(p) \wedge S_{Par_p} = EB)$ in γ.* By checking all actions of Algorithm $\mathcal{PIF}$, we can see that, as $S_q = EB$, EF-*action* is the only action that q may execute in $\gamma \mapsto \gamma'$. Now, $(\neg AbRoot(p) \Rightarrow GoodL(p)) \Rightarrow (L_p = L_q + 1)$ and $(L_p = L_q + 1 \wedge S_p \in \{B,F,P\}) \Rightarrow \neg EFeedback(q)$. So, EF-*action* is disabled at q in γ, q did not execute any action in $\gamma \mapsto \gamma'$, and as a consequence p is still enabled in γ', a contradiction.

□

Lemma 4.4 *If EF-action is enabled at p, it remains enabled until it is executed by p.*

Proof. Assume by contradiction that EF-*action* is enabled at p in γ and not in the next configuration γ' (*i.e.*, $\neg EFeedback(p)$ in γ'), but p did not execute EF-*action* in the step $\gamma \mapsto \gamma'$. Let $q \in Neig_p$ such that $Par_p = q$ ($p \neq r$). As EF-*action* is the enabled action at p with the highest priority (when, EF-*action* is enabled, EC- and EB-*actions* are disabled), p did not move in $\gamma \mapsto \gamma'$ and $S_p = EB$ in γ'. Now, $(\neg EFeedback(p) \wedge S_p = EB) \Rightarrow (\exists q \in Neig_p :: Par_q = p \wedge L_q > L_p \wedge S_q \notin \{EF,C\})$. So, there exists at least one neighbor of p, say q, that executes an action in $\gamma \mapsto \gamma'$ and that satisfies $(Par_q = p) \wedge (L_q > L_p) \wedge (S_q \notin \{EF,C\})$ in γ' (so that $\neg EFeedback(p)$ in γ'). There are two possibilities as discussed below:

- *q satisfies $(Par_q \neq p \vee L_q \leq L_p)$ in γ, but after executing an action, q satisfies $(Par_q = p \wedge L_q > L_p \wedge S_q \notin \{EF,C\})$ in γ'.* As $Par_q = p \wedge L_q > L_p$ in γ' and B-*action* is the only action that updates Variables Par_q or L_q, q executes B-*action* in $\gamma \mapsto \gamma'$. By B-*action*, q can only designate a process q' such that $S_{q'} = B$ in γ. Now, $S_p = EB$ in γ. So, $Par_q \neq p$ in γ', a contradiction.
- *q satisfies $(Par_q = p \wedge L_q > L_p \wedge S_q \in \{EF,C\})$ in γ, but after executing an action, q satisfies $(Par_q = p \wedge L_q > L_p \wedge S_q \notin \{EF,C\})$ in γ'.* If $S_q = C$ in γ, then q can only executes B-*action* in $\gamma \mapsto \gamma'$, and as in the previous case, $Par_q \neq p$ in γ', a contradiction. So, $S_q = EF$ in γ and EC-*action* is the only action that q may execute in $\gamma \mapsto \gamma'$. In this case, q still satisfies $(Par_q = p \wedge L_q > L_p \wedge S_q \in \{EF,C\})$ in γ' ($S_q := C$ in $\gamma \mapsto \gamma'$), a contradiction.

□

Lemma 4.5 *Let $p \in V$ such that $S_p \in \{EB,EF\}$. $S_p \in \{EB,EF\}$ holds (at least) until the tree of p is* `E-Dead`.

Proof. Consider a process p such that $S_p \in \{EB,EF\}$. By checking the actions of Algorithm 4.2, we can derive that when $S_p = EB$, the next action that p will execute is EF-$action$, *i.e.*, $S_p := EF$. Similarly, if $S_p = EF$, the next action that p will execute is EC-$action$, *i.e.*, $S_p := C$. Also, if p executes EC-$action$, $AbRoot(p)$ holds. Hence, p executes EC-$action$ only when it is the root of its abnormal tree, $S_p = EF$, and its abnormal tree, `Tree`(p), is `E-Dead` by Observation 4.5. □

Lemma 4.6 *All abnormal trees become* `E-Dead` *in at most $N-1$ rounds.*

Proof. Let $NotE_i$ be the set of processes p such that p is in an abnormal tree and $S_p \notin \{EB,EF\}$ at the first configuration of Round i. Let us define the function $\mathcal{F}$: $\mathbb{N} \to \mathbb{N}$ as follows:

$$\mathcal{F}(i) = \begin{cases} \infty & \textbf{if } NotE_i = \emptyset, \\ \min_{p \in NotE_i}(\{h(p)\}) & \textbf{otherwise}. \end{cases}$$

By Definition 4.8, $\mathcal{F}(i) = \infty$ (i.e., $NotE_i = \emptyset$) if and only if all abnormal trees are `E-Dead`. So, to prove the lemma, we have to show that $\mathcal{F}(N-1) = \infty$. We first show that if $NotE_i \neq \emptyset$, then $\mathcal{F}(i) < \mathcal{F}(i+1)$.

Assume by contradiction that there exists an execution satisfying $\mathcal{F}(i) \geq \mathcal{F}(i{+}1)$ while $NotE_i \neq \emptyset$ for a particular round i, *i.e.*, at the beginning of Round $i+1$, there is a process p in an abnormal tree such that $S_p \notin \{EB,EF\}$ and $h(p) \leq \mathcal{F}(i)$ (in particular, p is in an abnormal tree which is still not `E-Dead`). By Lemma 4.5, we know that any process q such that $S_q \in \{EB,EF\}$ satisfies $S_q \in \{EB,EF\}$ until its tree is `E-Dead`. So, two cases are possible for p.

a) *p attaches to an abnormal tree at the height $h \leq \mathcal{F}(i)$ during Round i (note that p may have left an abnormal tree before).* By Observations 4.2 and 4.3, p hooks on to an abnormal tree which is still not `E-Dead`. Also, from B-$action$, p attaches to an abnormal tree at the height h only if there exists a neighbor of p, say q, such that q is in an abnormal tree, $S_q = B$, and $h(q) \leq \mathcal{F}(i) - 1$. Now, by definition of $\mathcal{F}$ and Lemma 4.5, such a process q does not exist, a contradiction.

b) *p satisfies $(S_p \in \{B,F,P\}) \wedge (h(p) = \mathcal{F}(i))$ at the first configuration of Round i, p remains in its tree during i, and still satisfies $(S_p \in \{B,F,P\}) \wedge (h(p) = \mathcal{F}(i))$ at the first configuration of Round $i+1$.*

   1. If $h(p) = 0$, then, by Definitions 4.3 and 4.4, $p \neq r$ and $AbRoot(p)$: EB-$action$ is continuously enabled at p (Lemma 4.3). As the daemon is weakly fair, p sets S_p to EB before the end of the round and p satisfies $S_p \in \{EB,EF\}$ until its tree is `E-Dead` (Lemma 4.5), a contradiction.
   2. If $h(p) > 0$, then p satisfies $\neg AbRoot(p)$. Now, $(\neg AbRoot(p) \wedge S_p \notin \{EB,EF\}) \Rightarrow (S_{Par_p} \neq EF)$ and $(S_{Par_p} \neq EF \wedge h(Par_p) = \mathcal{F}(i) - 1) \Rightarrow (S_{Par_p} = EB)$. So, EB-$action$ is continuously enabled at p (Lemma 4.3). As the daemon is weakly fair, p sets S_p to EB before the end of the round and p satisfies $S_p \in \{EB,EF\}$ until its tree is `E-Dead` (Lemma 4.5), a contradiction.

Hence, if $NotE_i = \emptyset$, then $\mathcal{F}(i) < \mathcal{F}(i+1)$. Now, the maximal value of $\mathcal{F}(i)$ when $NotE_i \neq \emptyset$ is $N-2$, *i.e.*, the maximal height in any abnormal tree (all processes except the root can be in an abnormal tree). So, in the worst case, $\mathcal{F}$ returns ∞ from Round $i = N-1$, *i.e.*, all abnormal trees are `E-Dead` in at most $N-1$ rounds. □

Lemma 4.7 *The system contains no abnormal tree in at most $3N-3$ rounds.*

Proof. By Observation 4.5 and Lemma 4.6, each process p in abnormal trees satisfies `S-Trace(PPath`(p)`)` $= EB^*EF^*$ in at most $N-1$ rounds. Then no process can attach to these trees by Observations 4.2 and 4.3. We can also observe that no process p can leave any abnormal tree tree before $S_p = EF$ (only the root can leave its abnormal tree by EC-$action$ when any process q in its tree satisfies $S_q = EF$). Moreover, while there exists processes q in an abnormal tree such that $S_q = EB$, there exists (among the processes q) at least one process q' which has its EF-$action$ continuously enabled (by Observation 4.5 and Lemma 4.4). So, the worst case is obtained when any process of an abnormal tree satisfies $S = EB$. In this case, it is necessary to propagate the EF value from the leaves to the root. $H+1$ rounds are necessary for this propagation where H is the maximal height of the tree. Now, all processes except the root can be in an abnormal tree. This implies that the maximal height is $N-2$. Thus, in at most $N-1$

additional rounds, the system reaches a configuration γ where each process p in an abnormal tree satisfies $S_p = EF$. In γ, every process p satisfies one of the following cases: (i) $S_p = C$, (ii) $S_p = EF$ and p is in an abnormal tree, or (iii) $S_p \in \{B,F,P\}$ and p is in the normal tree. From γ, the EC-$action$ is continuously enabled at each root of each abnormal tree until all abnormal trees disappear (indeed, except the EC-$actions$, actions can be executed in the normal tree only). Hence, the EC-$actions$ will clean the successive abnormal roots until all abnormal trees disappear. In the worst case again, $N-1$ rounds are necessary to clean all abnormal trees. Hence, the system contains no abnormal trees in at most $3N-3$ rounds. □

The following lemma allows us to show that, starting from any configuration, the root can eventually start a PIF wave by B-$action$ (Theorem 4.2).

Lemma 4.8 *From any initial configuration containing no abnormal trees, r executes B-action in at most $6N$ rounds.*

Proof. Clearly, from such a configuration, the worst case is the following: The root satisfies $S_r = B$ and all the other processes have their S variable equal to C. Indeed, in this case, the system has to perform almost a complete PIF wave (a complete wave except the first step: B-$action$ of *r*). According to Algorithm $\mathcal{PIF}$, B-$action$ is then propagated to all processes in at most $N-1$ rounds. (Note that after executing B-$action$, a process is enabled to execute QR-$action$, so B-$actions$ and QR-$actions$ work in parallel.) After all processes executed their B-$action$, one extra round is necessary for the leaf processes of the broadcast to set their Que variable to R. Then, the W value is propagated into the Que variables by QW-$action$. The time used by the QW-$actions$ is bounded by the maximal height of the tree, *i.e.*., $N-1$ rounds (all processes, except the root, execute QW-$action$). By a similar reasoning taking in account that *r* also executes the respective actions, it is obvious that all the QA-$actions$, F-$actions$, P-$actions$, and C-$actions$ are successively propagated into the tree in at most N rounds. Hence, after $6N-1$ rounds, the system reaches a configuration where any process p satisfies $S_p = C$ and the root executes B-$action$ during the next round (more precisely, in the next step since *r* is the only enabled process). □

By Lemmas 4.7 and 4.8, the following result holds.

Theorem 4.2 *Starting from any initial configuration, r executes B-action in at most $9N-3$ rounds.*

We now conclude the proof by showing (in Theorem 4.3) that an execution starting from *r* executing a B-$action$ satisfies the PIF specification (Specification 4.1, page 10). To prove this, we use the following two technical lemmas.

Lemma 4.9 *Let p be a process in* `Tree`(r). *p can leave* `Tree`(r) *only when* `Tree`(r) *is* `Dead`.

Proof. By Definition 4.4, `Tree`(r) is a tree containing only processes p such that $(p = r) \vee \neg AbRoot(p)$. So, to leave `Tree`$(r)$, any process p must execute C-$action$, and as a consequence, p must be a leaf that satisfies $S_p = P$. Now, by Observation 4.4, if $S_p = P$, then any process q in `PPath`(p) satisfies $S_q = P$, particularly, $S_r = P$. Finally, by Observation 4.4 again, $S_r = P$ implies that any process in `Tree`(r) satisfies $S \in \{F,P\}$, which implies `Tree`(r) is `Dead`. □

Lemma 4.10 *Let p be a process in an abnormal tree such that $Que_p \in \{Q,R\}$. While p does not leave the tree, $Que_p \neq A$.*

Proof.

Assume that p also satisfies $AbRoot(p)$. If $S_p \in \{EB,EF\}$, then $S_p \in \{EB,EF\}$ holds until it leaves the tree (Lemma 4.5), so, QA-$action$ is disabled at p until it leaves the tree. If $S_p \in \{B,F,P\}$, then EB-$action$ is continuously enabled at p (Lemma 4.3). Now, EB-$action$ has a higher priority than QA-$action$. So, p cannot execute QA-$action$ before EB-$action$. After executing EB-$action$, $S_p = EB$ and we arrive at the previous case.

Assume now that p satisfies $\neg AbRoot(p)$. Two cases are then possible.

- $Que_p = R$. The R value is then propagated up in `PPath`(p). As $Que_q = R$, a process q in `PPath`(p) will modify Que_q only by QW-$action$ and only when $Que_{Par_q} = R$. Also, to switch Que_q to A, q must satisfy, in particular, $Que_q = W$ and $Que_{Par_q} = A$ (only *r* can switch from R to A). So, there always exists an R value in `PPath`(p) that provides such a process q to switch Que_q from W to A. Hence, R values are barriers for the A values, and while p is in the tree, p cannot receive any A.

- $Que_p = Q$. If p remains in the tree, then Que_p remains equal to Q until p executes QR-*action*. After executing QR-*action*, $Que_p = R$ and we arrive at the previous case.

□

Theorem 4.3 *From any configuration where r executes B-action, the execution satisfies Specification 4.1 (page 10).*

Proof. We first prove Property $[PIF1]$ of Specification 4.1, which is, every process receives any message *m* sent by *r* exactly once.

1. *Assume that there exists some process that never receives the message* m. Then, as the network is connected, there exists a process p which never receives *m*, but one of its neighbors, q, does. When q receives *m*, it executes B-*action*: q sets S_q to B and Que_q to Q.

   - *Assume that $S_p \neq C$.* If $Que_p \in \{W,A\}$, then (S_q,Que_q) will stay equal to (B,Q) until p executes QR-*action*. p eventually executes QR-*action*: $Que_p := R$. Now, since $S_p \neq C$ and p never receives *m*, we can deduce that p is in an abnormal tree and Que_p remains not equal to A until p leaves the tree by Lemma 4.10. So, until p leaves the tree, q cannot execute F-*action* (q does not satisfy $AnswerOK(q)$). By Lemma 4.7, p eventually leaves the tree, $S_p = C$ eventually holds.
   - *Assume that $S_p = C$ holds or eventually holds.* While $S_p = C$, q cannot execute F-*action* ($Feedback(q) \Rightarrow CFree(q)$). If p attaches to another abnormal tree again (B-*action*), then $(S_p,Que_p) := (B,Q)$ and $Que_p \neq A$ holds until p leaves the tree (Lemma 4.10). So, as before, q cannot execute F-*action*. Now, by Lemma 4.7, the system does not contain any abnormal tree in a finite time. So, eventually p continuously satisfies $S_p = C$ and $\forall p' \in Neig_p$, $(S_{p'} \neq C) \Rightarrow (Par_{p'} \neq p)$, *i.e.*, p continuously satisfies $Broadcast(p)$. As the daemon is weakly fair, p eventually receives *m* by executing B-*action*, a contradiction.

   Hence, each process receives *m* at least once.

2. *Assume that there exists a process* p *which receives the message* m *at least twice.* Clearly, $p \neq$ *r*, and according to the algorithm, p successively executes the B-, F-, P-, and C-*action* for *m* before it satisfies $Broadcast(p)$ for *m* again. After executing P-*action*, $\texttt{S-Trace}(\texttt{PPath}(p)) = P^*$ (Observation 4.4) particularly, $S_r = P$. By Observation 4.4 again, $S_r = P$ implies that any process q in $\texttt{Tree}(r)$ satisfies $\texttt{S-Trace}(\texttt{PPath}(q)) = P^*F^*$, and in this case, $\texttt{Tree}(r)$ is $\texttt{Dead}$. So, there is no process q such that $S_q = B$ for *m*. Hence, p cannot receive *m* for a second time, a contradiction.

We now show Property $[PIF2]$. First, by Case 1 above, we know that each process attaches to the normal tree (by B-*action*) during the broadcast of *m*. Also, by Lemma 4.9, after attaching to $\texttt{Tree}(r)$, the processes cannot leave the tree before it is $\texttt{Dead}$. Now, by Lemma 4.2, *r* eventually executes B-*action* again, and *r* executes B-*action* only if $S_r = C$. So, $\texttt{Tree}(r)$ is eventually empty, and $\texttt{Tree}(r)$ is $\texttt{Dead}$. By Observation 4.4, $\texttt{Tree}(r)$ is $\texttt{Dead}$ if and only if any process p in $\texttt{Tree}(r)$ satisfies $\texttt{S-Trace}(\texttt{PPath}(p)) = P^*F^*$. Also, *r* changes S_r to P only when $S_r = F$. So, the system eventually reaches a configuration where any process p is in $\texttt{Tree}(r)$ and satisfies $S_p = F$. *r* eventually receives an acknowledgment from every non-root process.

Hence, after *r* executes B-*action*, the execution satisfies Specification 4.1. □

By Remark 4.1 (page 10), Theorems 4.2, and 4.3, the following theorem is obvious.

Theorem 4.4 *Algorithm $\mathcal{PIF}$ is snap-stabilizing for Specification 4.1 under a distributed weakly fair daemon.*

4.2.3 Proof assuming an Unfair Daemon

To complete the proof of snap-stabilization of Algorithm $\mathcal{PIF}$ under a distributed unfair daemon, we need to show that each PIF Wave is finite in terms of steps. We first show that the abnormal trees can only generate a finite number of actions during the whole execution.

Lemma 4.11 *Any non-root process p that attaches to an abnormal tree (by B-action) cannot execute F-action before leaving the tree.*

Proof. When attaching to an abnormal tree, any process p sets (S_p,Que_p) to (B,Q). From that point on, $Que_p \neq A$ holds until p leaves the tree (Lemma 4.10). Hence, while p is in the tree, $\neg AnswerOk(p)$ holds, and F-$action$ is disabled at p. □

From Lemma 4.11, we can deduce the following corollary:

Corollary 4.1 *During the whole execution, any non-root process p cannot execute F-$action$ more than once while it is not in* $\texttt{Tree}(r)$.

To execute P- or C-$action$ while a process is in an abnormal tree the second time, it must execute an F-$action$ before.

Corollary 4.2 *During the whole execution, any non-root process p cannot execute P-$action$ or C-$action$ more than once while it is not in* $\texttt{Tree}(r)$.

Lemma 4.12 *After attaching to an abnormal tree (by B-action), a non-root process leaves the tree only when it is* Dead.

Proof. Two actions allow p to leave the tree: C-$action$ and EC-$action$. So, we prove the lemma with the two following claims:

- *p cannot execute C-$action$.* After attaching to the tree, p cannot execute F-$action$ until it leaves the tree (Lemma 4.11). So, while p is in the tree, $S_p \neq F$, and p cannot execute P-$action$. Hence, while p is in the tree, $S_p \neq P$, and C-$action$ is disabled at p.
- *p executes EC-$action$ only if its tree is* Dead. If p executes EC-$action$, then p satisfies $(S_p = EF) \wedge AbRoot(p)$. So, p is the root of its abnormal tree, $\texttt{Tree}(p)$. $S_p = EF$ implies that any process q in $\texttt{Tree}(p)$ also satisfies $S_q = EF$ by Observation 4.4: $\texttt{Tree}(p)$ is Dead. Hence, p execute EC-$action$ only if its tree is Dead.

□

By Lemma 4.12 and Observation 4.2, the next lemma follows:

Lemma 4.13 *During the whole execution, any non-root process q attaches to the abnormal tree rooted at p (p ≠ r) at most once.*

Lemma 4.14 *During the whole execution, O(N^2) attaching actions (B-action) are executed in the abnormal trees.*

Proof. By Lemma 4.13, we know that each non-root process ($N-1$ of them) can execute B-$action$ to attach at most once to each abnormal tree. Then the system may contain at most $N-1$ abnormal trees. Hence, $O(N^2)$ B-$actions$ can be executed in the abnormal trees. □

By Lemma 4.14, and Corollaries 4.1 and 4.2, we obtain the following result.

Lemma 4.15 *During the whole execution, the abnormal trees generate an overhead of O(N^2) actions of the* PIF Part.

We now focus on the *Question Part*.

Lemma 4.16 *Let p be a non-root process such that $AbRoot(p)$. While $AbRoot(p)$, p cannot execute any action of the* Question Part.

Proof. Assume by contradiction that p executes an action of the *Question Part* while satisfying $AbRoot(p)$. In this case, p also satisfies $S_p \in \{B,F\}$. Then, $(S_p \in \{B,F\} \wedge AbRoot(p)) \Rightarrow EBroadcast(p)$, and as EB-$action$ has an higher priority than any action of *Question Part*, if p moves, then p executes EB-$action$, a contradiction. □

Lemma 4.17 *Each B-action generates O($\Delta \times N$) actions of the* Question Part.

Proof. By executing B-$action$, a process p sets (S_p,Que_p) to (B,Q). Then, Que_p remains equal to Q until the following condition holds: $\forall q \in Neig_p$, $(S_q \neq C) \Rightarrow (Que_q \in \{Q,R\})$. If a neighbor of p, say q, executes QR-$action$, then $S_q \neq C$ and q satisfies $Que_q = R$ until p executes $Que_p := R$ (QR-$action$). Finally, when each neighbor q of p satisfies $(S_q \neq C) \Rightarrow (Que_q \in \{Q,R\})$, p can execute QR-$action$. So, each B-$action$ generates at most $\Delta + 1$ new R values in the system.

In the worst case, these R values ($O(\Delta)$) are then propagated up in (and only in) the `PPath` of p and its neighbors q such that $S_q \neq C$ (using QR-$action$) until a process v such that $(v = r) \vee AbRoot(Par_v)$ (Predicate $Reset$ and Lemma 4.16). Now, the number of processes in each `PPath` is bounded by N. So, the cost of these propagations is in $O(\Delta \times N)$ actions.

In the worst case again, p and all its neighbors q such that $S_q \neq C$ ($O(\Delta)$) execute $Que := W$. Also in the worst case, the generated W values are propagated up in (and only in) the `PPath` of p and its neighbors q such that $S_q \neq C$ (using QW-$action$) until a process v such that $(v = r) \vee AbRoot(Par_v)$ (Predicate $Wait$ and Lemma 4.16). So, similar to the R values, the cost of these propagations is $O(\Delta \times N)$ actions.

Finally, in the worst case, each of the $O(\Delta)$ W values previously generated is propagated into the normal tree. In this case, each W value may generate a broadcast of an A value into the whole normal tree. (r is the only process able to generate a new A value). So, once again the cost of these broadcasts is $O(\Delta \times N)$ actions.

Hence, each B-$action$ generates $O(\Delta \times N)$ actions of the *Question Part*. □

Lemma 4.18 *During the whole execution, the abnormal trees generate an overhead of $O(\Delta \times N^3)$ actions of the* Question Part*.*

Proof. A process propagates a question in trees (*i.e.*, actions of the *Question Part*) due to the initial configuration or when it attaches to a tree (B-$action$). Each time it attaches to an abnormal tree, it generates $O(\Delta \times N)$ actions of the *Question Part* (Lemma 4.17). Note that the number of actions of the *Question Part* generated if it is in an abnormal tree since the initial configuration, is of the same order ($O(\Delta \times N)$). Then, $O(N)$ processes are in abnormal trees in the initial configuration, and $O(N^2)$ processes attach to abnormal trees in the execution (Lemma 4.14). Thus, abnormal trees generate $O(\Delta \times N^3)$ actions of the *Question Part*. □

Finally, we evaluate the overhead due to the *Correction Part*.

Lemma 4.19 *During the whole execution, $O(N^2)$ actions of the* Correction Part *are executed to delete the abnormal trees.*

Proof. In the worst case, each process in abnormal trees has to successively execute the three actions of the *Correction Part* (*i.e.*, EB-, EF-, and EC-$action$) in order to leave its tree. So, any process leaves an abnormal tree in $O(1)$ actions of the *Correction Part*. Now, $O(N)$ processes are in abnormal trees at the initial configuration and $O(N^2)$ processes attach to abnormal trees during the whole execution (Lemma 4.14). Hence, the lemma is trivially verified. □

From Lemmas 4.15, 4.18, and 4.19, we can deduce the following result:

Lemma 4.20 *During the whole execution, the abnormal trees generate an overhead of $O(\Delta \times N^3)$ actions before getting deleted.*

We now show that, from any configuration, `Tree`(r) (the normal tree) can only generate a finite number of actions before the root starts a PIF wave (by B-$action$). As the *correction* part is not used in the normal tree, it is not considered in the following.

By Lemma 4.9 (page 19) and Observation 4.2, we get the following result:

Lemma 4.21 *From any configuration, each non-root process attaches to* `Tree`(r) *(by B-$action$) at most once before r executes B-$action$.*

Lemma 4.22 *From any configuration, before r executes B-$action$, each process executes F-$action$ at most once while it is in* `Tree`(r).

Proof. Assume by contradiction that before *r* execute $B\text{-}action$, a process p executes $F\text{-}action$ at least twice while it is in `Tree`(r). After the first execution of $F\text{-}action$, p satisfies $S_p = F$. Then, according to the algorithm, p must successively executes the P-, C-, and $B\text{-}action$ before it executes $F\text{-}action$ again. Then, any process q in `Tree`(r) satisfies $S_q \neq B$ when p executes $C\text{-}action$ (`Tree`(r) is `Dead` by Lemma 4.9). Hence, *r* must initiate another broadcast by $B\text{-}action$ so that p attaches to `Tree`(r) by $B\text{-}action$, a contradiction. □

Using a reasoning similar to the one used for Corollary 4.2, we can deduce the following corollary from Lemma 4.22:

Corollary 4.3 *From any configuration, before r executes B-action, each process executes P-action and C-action at most once while it is in* `Tree`(r)*.*

By Lemmas 4.21, 4.22, and Corollary 4.3, we get the following:

Lemma 4.23 *From any configuration, O(N) actions of the* PIF Part *are executed in* `Tree`(r) *before r executes B-action.*

Lemma 4.24 *From any configuration,* `Tree`(r) *generates O($\Delta \times N^2$) actions of the* Question Part *before r executes B-action.*

Proof. Similar to the proof of Lemma 4.18, and Lemmas 4.17 and 4.21, it is easy to see that `Tree`(r) generates $O(\Delta \times N^2)$ actions of the *Question Part* before *r* executes $B\text{-}action$. □

By Lemma 4.23 and 4.24, we obtain the following result:

Lemma 4.25 *From any configuration,* `Tree`(r) *generates O($\Delta \times N^2$) actions before r executes B-action.*

By Lemmas 4.20 and 4.25, we can claim the following:

Theorem 4.5 *From any configuration, r executes B-action in O($\Delta \times N^3$) steps.*

A complete PIF wave is executed between two executions of $B\text{-}action$ at *r*.

Corollary 4.4 *From any configuration, a wave of Algorithm $\mathcal{PIF}$ is executed in O($\Delta \times N^3$) steps.*

By Theorem 4.4 and Corollary 4.4, we obtain the following result:

Theorem 4.6 *Algorithm $\mathcal{PIF}$ is snap-stabilizing for Specification 4.1 under a distributed unfair daemon.*

The following additional property of Algorithm $\mathcal{PIF}$ will be used later to design a snap-stabilizing reset algorithm. It can be deduced from the proof of Property $[PIF2]$ in Theorem 4.3 (page 20) and Theorem 4.6.

Property 4.1 *After r starts to broadcast a message* m*, the system reaches a configuration in a finite number of rounds (resp., steps) where all processes have acknowledged the receipt of* m*.*

4.3 Complexity Analysis

We first consider the complexity in terms of steps:

Delay: By Theorem 4.5, we can state that the delay of Algorithm $\mathcal{PIF}$ is $O(\Delta \times N^3)$ steps.

Complete PIF wave: By Corollary 4.4, Algorithm $\mathcal{PIF}$ executes a complete PIF wave in $O(\Delta \times N^3)$ steps.

We now consider the complexity in terms of rounds:

Delay: By Theorems 4.2 (page 19), we can state that the delay of Algorithm $\mathcal{PIF}$ is at most $9N - 3$ rounds.

Complete PIF wave: We can deduce from Theorem 4.2, and Lemmas 4.7, and 4.8 (pages 18 and 19) that Algorithm $\mathcal{PIF}$ executes a complete PIF wave in at most $9N - 3 + 6N = 15N - 3$ rounds.

Hence, the round complexities of Algorithm $\mathcal{PIF}$ asymptotically match those of the previous solutions ([10, 11]) using the same memory requirement, namely $O(\log N)$ bits per process. Furthermore and contrary to the previous solutions, Algorithm $\mathcal{PIF}$ has bounded step complexities due to the fact that it works under an unfair daemon.

5 Other Key Algorithms

In this section, we present four essential snap-stabilizing algorithms to develop a universal transformer (see Section 6). These four algorithms are based on the snap-stabilizing PIF algorithm presented in previous sections.

5.1 Snap-Stabilizing Leader Election

A *leader election* algorithm ensures that upon termination of the algorithm, exactly one process is distinguished as the *leader* among all the processes of the network. Our purpose here is not to propose a very efficient leader election algorithm, but rather to show that we can design a snap-stabilizing leader election algorithm based on Algorithm $\mathcal{PIF}$. Below we describe a basic solution.

We first design a snap-stabilizing maximum ID computation using Algorithm $\mathcal{PIF}$. In the following, we refer to this algorithm as Algorithm $\mathcal{M}$. Every process in the network runs Algorithm $\mathcal{M}$ in parallel. Some processes may not initiate their own version of Algorithm $\mathcal{M}$ on their own. However, upon receiving an $\mathcal{M}$ message from some other process, they also initiate the same algorithm. When the PIF waves corresponding to Algorithm $\mathcal{M}$ terminate, all processes have elected the same ID corresponding to the leader.

In this paper, we use a weaker version of the leader election in the following sense: Each process just checks if it is the actual leader (using $\mathcal{M}$) only when it needs to. In the rest of the paper, we refer to this algorithm as $\mathcal{LE}$.

5.2 Snap-Stabilizing Reset

The *reset* algorithm is used in a faulty environment to "reset" the system to a pre-defined "good" configuration for a problem $\mathcal{P}$. A good global configuration can be restricted to a normal initial configuration (see Section 3) to simplify the design process. Any normal configuration can be used as a good configuration. Our approach to design the reset algorithm is similar to the one in [35].

We restrict the reset algorithm to be initiated by one and only one initiator, called i. We call this rooted reset algorithm $\mathcal{R}_i^{\mathcal{P}}$. i uses Algorithm $\mathcal{PIF}$ as follows: (a) i initiates the broadcast of an "abort" message and stops its local execution of $\mathcal{P}$. (b) Upon receiving the message, the non-initiators also abort their local execution of $\mathcal{P}$. (c) All the processes (including i) reset their variables related to the problem $\mathcal{P}$ when they feedback. (d) Finally, the processes can roll back their local execution of $\mathcal{P}$ only when they stop participating in the PIF wave corresponding to $\mathcal{R}_i^{\mathcal{P}}$. Since Algorithm $\mathcal{PIF}$ is snap-stabilizing, all the processes will receive the abort message. From Property 4.1 (page 23), after starting $\mathcal{R}_i^{\mathcal{P}}$, the system reaches, in a finite time, a configuration where all processes have their variables reset. Hence, after i stops participating in the PIF wave, the system is guaranteed to be in a normal initial configuration.

5.3 Snap-Stabilizing Snapshot

The problem of distributed *snapshot* is quite challenging in a faulty environment. The problem is to collect some data about the system. This data must be adequate to verify the coherence of the system. The consistent snapshot consists in a collection of local states of all processes such that this collection can correspond to some normal configuration of the problem.

Our solution, noted $\mathcal{S}^{\mathcal{P}}$, is an adaptation in our model (locally shared memory) of the snapshot algorithm provided in [36] (written in the message-passing model). So, before we present our algorithm, we briefly describe the main idea of the snapshot algorithm in [36] applied on a problem $\mathcal{P}$. In this algorithm, a process initiates the snapshot by first recording the state of $\mathcal{P}$ and then broadcasting a marker to all its neighbors. Upon receiving the first marker message, a process records its own state of $\mathcal{P}$ and sends the marker to all its neighbors (including the sender of the marker). A marked process records every message received from a neighbor which did not send the marker yet. When a process i receives a marker from all its neighbors, i completes its participation in the snapshot algorithm by sending a report (recorded state and messages) to the initiator. In our model (locally shared memory), the messages are modeled by the ability of a process to read the state of its neighbors. So, the collection of the recorded messages at i from a neighbor j (in the message-passing model) is replaced by the history of change of state of j after the marking event of p and before that of q.

The aim of Algorithm $\mathcal{S}^{\mathcal{P}}$ is to gather at the initiator the histories of local states of each process from the beginning to the end of its participation to the snapshot (actually a PIF wave). In our solution, we use a local stack at each process to store its history. Upon receiving a broadcast message, a process p resets its stack to its current local configuration.

This local configuration is constituted by the state of p and its neighbors (the variables related to the problem $\mathcal{P}$ plus the PIF variables). Then, at each move, the new local configuration is pushed onto the stack until the feedback. During the feedback phase, the process pushes its current configuration on the stack for the last time and merges its own report (actually its stack) and that of its children (in the tree built during the broadcast phase) in a report variable. As Algorithm $\mathcal{PIF}$ ensures that any process that executes the broadcast and the feedback phase, the report variable of the initiator includes the report of that process at the end of the feedback phase. At least as much data as the snapshot algorithm of [36] are saved during the execution of Algorithm $\mathcal{S}^{\mathcal{P}}$. Hence, we obtain a consistent snapshot when Algorithm $\mathcal{S}^{\mathcal{P}}$ terminates.

5.4 Snap-Stabilizing Termination Detection

The *termination detection* can be (may not be efficiently) performed using any snapshot algorithm. Using $\mathcal{S}^{\mathcal{P}}$, our termination detection algorithm just needs to check the reports at the end of any snapshot. For any snapshot started after the termination of the algorithm $\mathcal{A}$, the report of any process consists of two identical configurations: the current local configurations at the broadcast and the feedback, in that order. In the following, we denote by $\mathcal{TD}^{\mathcal{A}}$ the termination detection algorithm for the algorithm $\mathcal{A}$.

6 Transformer

Let $\mathcal{A}$ be an algorithm designed for any identified network of arbitrary topology (note that $\mathcal{A}$ is neither self- nor snap-stabilizing). In this section, we show how to transform $\mathcal{A}$ into a snap-stabilizing algorithm.

First, we define some composition definitions used in this section. A *fair parallel composition* of two algorithms A and B is an algorithm denoted by $A||B$, where actions of A (resp., B) occur infinitely often in any execution if an infinite number of configurations contains enabled actions of A (resp., B). A *sequential composition* of two algorithms A and B is the algorithm denoted by $A \to B$, where B can start if and only if A has terminated. The starting of B can be also dependent on some input predicate c: $A \underset{c}{\to} B$ means that B can start if and only if A has terminated and c is true.

Remark 6.1 *As in [28], we assume that:*

1. *We can define for $\mathcal{A}$ a predicate* `OK` *which characterizes the normal configurations of the system. This predicate can be computed by a snapshot.*
2. *If $\mathcal{A}$ solves an infinite problem, then the problem can be defined by a* suffix-closed *specification.*

Moreover, we assume that each process knows an upper bound on the number of processes in the network.

By Remark 6.1 (Part 2), every execution of $\mathcal{A}$ can be seen as an infinite repetition of *finite sub-executions*. Every sub-execution is initiated using a starting action. Any starting action is triggered upon an external (*w.r.t.* the algorithm) request only. For example, the depth-first token circulation can be seen as successive depth-first traversals of the network, each of them starts upon receiving an external request.

We now present the ideas of the transformer by considering two classes of algorithms:

(a) The algorithms that have a unique initiator (the algorithm is associated to a process and cannot be merged with a similar algorithm initiated by another process).

(b) The algorithms that can be initiated by several processes.

6.1 Single Initiator Algorithm

Consider an algorithm $\mathcal{A}$ which is from Class (a) with Process i as initiator. A simple attempt to transform $\mathcal{A}$ into a snap-stabilizing algorithm is to reset the network using $\mathcal{R}_i^{\mathcal{A}}$ before starting $\mathcal{A}$. So, we add $\mathcal{R}_i^{\mathcal{A}}$ as an header of $\mathcal{A}$ such that $\mathcal{A}$ starts only after $\mathcal{R}_i^{\mathcal{A}}$ terminates, *i.e.*, we consider the algorithm $\mathcal{R}_i^{\mathcal{A}} \to \mathcal{A}$ and the starting action of $\mathcal{R}_i^{\mathcal{A}} \to \mathcal{A}$ is the one of $\mathcal{R}_i^{\mathcal{A}}$. Thus, starting $\mathcal{A}$ now implies first running $\mathcal{R}_i^{\mathcal{A}}$ and then $\mathcal{A}$. As the system has been reset for $\mathcal{A}$ in a snap-stabilizing manner, obviously, $\mathcal{A}$ behaves as in a non-faulty situation, *i.e.*, according to its specification.

Assume now that no starting action of $\mathcal{R}_i^{\mathcal{A}}$ was executed after a fault occurred, but $\mathcal{R}_i^{\mathcal{A}} \to \mathcal{A}$ is still running. Then, there is no guarantee that $\mathcal{R}_i^{\mathcal{A}} \to \mathcal{A}$ terminates. Hence, we cannot ensure that the execution will contain at least one starting action of $\mathcal{R}_i^{\mathcal{A}} \to \mathcal{A}$. Since $\mathcal{R}_i^{\mathcal{A}}$ is snap-stabilizing, we know that $\mathcal{R}_i^{\mathcal{A}}$ eventually terminates. So, the reason why $\mathcal{R}_i^{\mathcal{A}} \to \mathcal{A}$ may not terminate is due to the fact that $\mathcal{A}$ may not terminate (recall that $\mathcal{A}$ is not even self-stabilizing). To prevent the system from this kind of deadlock or livelock, a checking procedure must be added to detect if the system is not in a normal configuration and, in this case, abort the current execution before starting a new one. The checking procedure consists of taking snapshot of the system regularly (using the snap-stabilizing snapshot algorithm $\mathcal{S}_i^{\mathcal{A}}$). We use the snapshots to compute two predicates: $\mathtt{OK}$ which characterizes the normal configurations of the system and $\mathtt{TD}$ which determines if $\mathcal{A}$ is terminated. As explained in Subsection 5.4, we can use Algorithm $\mathcal{S}_i^{\mathcal{A}}$ for termination detection. Process i now waits until $\mathcal{S}_i^{\mathcal{A}}$ returns either $\neg\mathtt{OK}$ or $\mathtt{TD}$, before starting $\mathcal{R}_i^{\mathcal{A}} \to \mathcal{A}$.

In order to design the checking procedure without fairness assumptions (*i.e.,* under an unfair daemon), we must not execute snapshots concurrently with $\mathcal{A}$. Otherwise, an unbounded number of steps of $\mathcal{A}$ can be executed before the completion of the first snapshot. Hence, we propose to schedule at most one step of $\mathcal{A}$ per process, using a $\mathcal{PIF}$ wave, before executing a snapshot of the system. After that, if the configuration is a normal one (*i.e.*, if $\mathcal{S}_i^{\mathcal{A}}$ returns $\mathtt{OK}$), we repeat the procedure until $\mathcal{A}$ terminates (*i.e.*, until $\mathcal{S}_i^{\mathcal{A}}$ returns $\mathtt{TD}$). Otherwise, we abort the current execution so that i can start $\mathcal{R}_i^{\mathcal{A}} \to \mathcal{A}$. Hence, the checking procedure ensures that $\mathcal{R}_i^{\mathcal{A}} \to \mathcal{A}$ eventually starts, but without risking to abort any normal execution of $\mathcal{A}$.

The checking procedure is designed as follows: $\mathcal{PIF}_i^{\mathcal{A}}$ waves and $\mathcal{S}_i^{\mathcal{A}}$ waves are performed in sequence until the predicate $\neg\mathtt{OK} \vee \mathtt{TD}$ is satisfied (at i). During each wave of $\mathcal{PIF}_i^{\mathcal{A}}$, any process can execute at most one action of $\mathcal{A}$: upon receiving a broadcast wave. Since $\neg\mathtt{OK} \vee \mathtt{TD}$ is satisfied, i can start $\mathcal{R}_i^{\mathcal{A}} \to \mathcal{A}$ (upon a request $\mathtt{Req}_i$) without risking to abort a normal execution of $\mathcal{A}$. Indeed, either the behavior is fuzzy ($\neg\mathtt{OK}$) or the previous execution of $\mathcal{A}$ is terminated ($\mathtt{TD}$).

Hence, we obtain a new algorithm, $\mathtt{TRANS1}_{\mathcal{A}}$, which is shown in Figure 6.1.

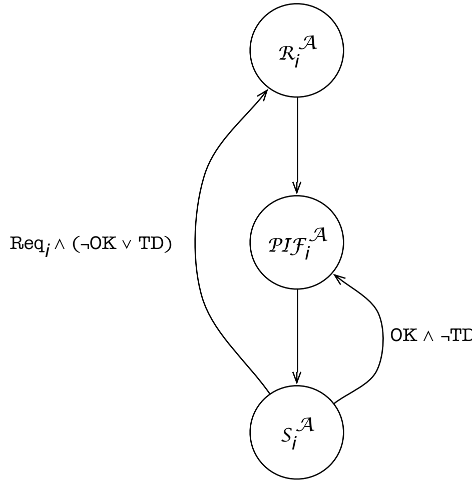

Figure 6.1: $\mathtt{TRANS1}_{\mathcal{A}}$

The projection upon the variables of $\mathcal{R}_i^{\mathcal{A}} \to \mathcal{A}$ of any execution of $\mathtt{TRANS1}_{\mathcal{A}}$ can be decomposed as a default prefix followed by complete executions of $\mathcal{R}_i^{\mathcal{A}} \to \mathcal{A}$. The default prefix is:

- Either, a suffix of $\mathcal{R}_i^{\mathcal{A}} \to \mathcal{A}$ if $\mathcal{S}_i^{\mathcal{A}}$ always returns $\mathtt{OK}$,
- Or, any fuzzy behavior of $\mathcal{R}_i^{\mathcal{A}} \to \mathcal{A}$ (without any starting action) stopped by $\mathcal{S}_i^{\mathcal{A}}$ returning $\neg\mathtt{OK}$.

Finally, as any execution of $\mathtt{TRANS1}_{\mathcal{A}}$ just consists of sequences of $\mathcal{PIF}$ waves, it is guaranteed that any execution of $\mathtt{TRANS1}_{\mathcal{A}}$ is finite in terms of steps even if the daemon is assumed to be unfair. Hence, the following theorem immediately follows:

Theorem 6.1 *Let $\mathcal{A}$ be a distributed algorithm which has a unique initiator. Then, the transformed algorithm $\mathtt{TRANS1}_{\mathcal{A}}$ is snap-stabilizing for the specification of the problem solved by $\mathcal{A}$ under a distributed unfair daemon.*

Implementation in the locally shared memory model Before presenting how to implement $\mathtt{TRANS1}_{\mathcal{A}}$, let us define the predicate $\mathcal{X}.End(p)$. We have already seen in Section 4 that any process p satisfies $S_p = C$ in Algorithm $\mathcal{PIF}$ when p is waiting for the next broadcast message, *i.e.*, when it does not participate in any PIF wave. Let $\mathcal{X}.End(p) \equiv (S_p = C)$ be a predicate for a process p in the specific PIF-based algorithm $\mathcal{X}$. In particular, when $\mathcal{X}.End(i)$ is satisfied, any PIF wave of $\mathcal{X}$ started by i (if any) is now terminated.

In the discussion above Theorem 6.1, we have seen that $\mathcal{R}_i^{\mathcal{A}}$ must not start before a snap-stabilizing snapshot (*i.e.*, $\mathcal{S}_i^{\mathcal{A}}$) returns $\neg\mathtt{OK} \vee \mathtt{TD}$. So, we have to add the condition $\mathcal{S}_i^{\mathcal{A}}.End(i) \wedge (\neg\mathtt{OK} \vee \mathtt{TD})$ into the guard of the starting action of $\mathcal{R}_i^{\mathcal{A}}$, meaning that $\mathcal{R}_i^{\mathcal{A}}$ can start only when $\mathcal{S}_i^{\mathcal{A}}$ terminates and outputs $(\neg\mathtt{OK} \vee \mathtt{TD})$.

During its participation in a reset, any process p (including i) must stop its local executions of $\mathcal{PIF}_i^{\mathcal{A}}$ (following the explanation given in Subsection 5.2) and $\mathcal{S}_i^{\mathcal{A}}$ (if a snapshot is done concurrently with a reset, the snapshot may incorrectly output $\neg\mathtt{OK}$). For that reason, we add the condition $\mathcal{R}_i^{\mathcal{A}}.End(p)$ in the guard of each action of each process p (including i) in Algorithms $\mathcal{PIF}_i^{\mathcal{A}}$ and $\mathcal{S}_i^{\mathcal{A}}$.

Until $\mathcal{S}_i^{\mathcal{A}}.End(i) \wedge (\neg\mathtt{OK} \vee \mathtt{TD})$ is satisfied, waves of $\mathcal{PIF}_i^{\mathcal{A}}$ and $\mathcal{S}_i^{\mathcal{A}}$ must be executed in sequence. For this, we use an additional variable $Turn_i \in \{1, 2\}$. This value determines the next wave to be performed: $Turn_i = 1$ (resp. $Turn_i = 2$) means that this is the turn of $\mathcal{PIF}_i^{\mathcal{A}}$ (resp. $\mathcal{S}_i^{\mathcal{A}}$). Hence, $Turn_i$ is set to 1 at the termination of each wave of $\mathcal{R}_i^{\mathcal{A}}$ and $\mathcal{S}_i^{\mathcal{A}}$. Conversely, $Turn_i$ is set to 2 at the end of each $\mathcal{PIF}_i^{\mathcal{A}}$ wave. Then, a $\mathcal{PIF}_i^{\mathcal{A}}$ can start only if $\mathcal{S}_i^{\mathcal{A}}.End(i) \wedge (\mathtt{OK} \wedge \neg\mathtt{TD}) \wedge (Turn_i = 1)$, meaning that $\mathcal{PIF}_i^{\mathcal{A}}$ can start only when $\mathcal{S}_i^{\mathcal{A}}$ terminates, outputs $(\mathtt{OK} \vee \neg\mathtt{TD})$, and this is the turn of $\mathcal{PIF}_i^{\mathcal{A}}$. Thus, this condition must be added in the guard of the starting action of $\mathcal{PIF}_i^{\mathcal{A}}$. For similar reasons, the condition $\mathcal{PIF}_i^{\mathcal{A}}.End(i) \wedge (\mathtt{OK} \wedge \neg\mathtt{TD}) \wedge (Turn_i = 2)$ must be added in the guard of the starting action of $\mathcal{S}_i^{\mathcal{A}}$.

Finally, each process p can execute an action of $\mathcal{A}$ only when switching to the broadcast phase in $\mathcal{PIF}_i^{\mathcal{A}}$. To apply this mechanism, each action of p in $\mathcal{A}$ needs to be moved into the statement of its broadcast action in $\mathcal{PIF}_i^{\mathcal{A}}$ (the guard of each action being replaced by the corresponding *if* statement).

6.2 Multi-Initiator Algorithm

Let $\mathcal{A}$ be a distributed algorithm of Class (b), p be a process, and e be a normal execution of $\mathcal{A}$. During e, p should be eventually enabled to start $\mathcal{A}$ using a *starting action* upon receiving a request. Moreover, at that time the starting action should be continuously enabled until p is activated by the daemon. Otherwise, the liveness part of the specification of $\mathcal{A}$ can be violated in a safe environment.

Remark 6.2 *After the system is in a normal configuration, upon a receiving a request, any initiator will be eventually enabled continuously to start $\mathcal{A}$.*

We now propose to transform $\mathcal{A}$ into a snap-stabilizing algorithm working under a *weakly fair* daemon. We then explain how to make the transformed algorithm working under an *unfair* daemon.

Unlike the single-initiator case, an initiator cannot blindly reset the system before starting $\mathcal{A}$. Indeed, such a reset may interrupt an execution started by some other initiator. The idea here is to let execute $\mathcal{A}$ by all processes to prevent deadlocks. However, we should carefully address the execution of starting actions. So, we precede the initialization of a process i in $\mathcal{A}$ by the execution of the header $\mathcal{H}_i^{\mathcal{A}}$. As in Class (a), the initialization of the transformed algorithm is the one of $\mathcal{H}_i^{\mathcal{A}}$. More precisely, upon receiving a request, a process i waits until it is able to start $\mathcal{A}$. Then, it first initializes $\mathcal{H}_i^{\mathcal{A}}$, then delays the initialization of $\mathcal{A}$ until the completion of $\mathcal{H}_i^{\mathcal{A}}$.

PROTOCOL $\mathcal{H}_i^{\mathcal{A}}$
(1) IF $\mathcal{LE}_i$ RETURNS i THEN /* i is the actual leader */
(2) $\quad\quad \mathcal{L}_i^{\mathcal{A}}$
(3) ELSE
(4) $\quad\quad \mathcal{RI}_i^{\mathcal{A}}$
(5) ENDIF

The goal of the header $\mathcal{H}_i^{\mathcal{A}}$ is to check if the system is in a normal configuration. If not, the system is reset. So, at the termination of $\mathcal{H}_i^{\mathcal{A}}$, the system is in a normal configuration, and i can start $\mathcal{A}$.

The snapshot and the reset of the system must be exclusively executed by a single process. So, the first part of $\mathcal{H}_i^{\mathcal{A}}$ consists of waking up the leader so that it performs these two algorithms. Process i first initiates a snap-stabilizing leader election $\mathcal{LE}_i$. At the termination of $\mathcal{LE}_i$, the following two cases are possible:

1. i is the leader of the network. Then, i executes Algorithm $\mathcal{L}_i^{\mathcal{A}}$ (see Line 2 in $\mathcal{H}_i^{\mathcal{A}}$). This algorithm ensures that the leader (here i) will execute a snapshot *alone*, possibly followed by a reset, using $\mathcal{CR}_i^{\mathcal{A}}$.

2. i is not the leader of the network. Then, i requests the leader of the network the permission to initiate $\mathcal{A}$. To send the request, we use $\mathcal{RI}_i^{\mathcal{A}}$ (based on $\mathcal{PIF}$) (see Line 4 in $\mathcal{H}_i^{\mathcal{A}}$). Process i initiates the broadcast of a "Request for Initialization of $\mathcal{A}$" message, then waits for the end of the feedback phase.

   Note that a process j is able to receive broadcasts of $\mathcal{RI}^{\mathcal{A}}$ initiated by some other processes if and only if it satisfies $\texttt{Brd}_j \equiv (\exists k \in Neig_j, \mathcal{RI}_k^{\mathcal{A}}.S = B)$.

   As Algorithm $\mathcal{RI}_i^{\mathcal{A}}$ is snap-stabilizing, the request will be received by every process, in particular the actual leader of the network. Let ℓ be this leader. Upon receipt of a "Request for Initialization of $\mathcal{A}$" from i (*i.e.*, when $\texttt{Brd}_j$ is satisfied), any process j launches $\mathcal{L}_j^{\mathcal{A}}$. j waits for the termination of $\mathcal{L}_j^{\mathcal{A}}$ before executing its feedback phase. However, no process but ℓ will go further than the leader control (see Line 1 in Algorithm $\mathcal{L}^{\mathcal{A}}$). So, during the feedback phase of $\mathcal{RI}_i^{\mathcal{A}}$, ℓ will be the only process able to stop the progression of the feedbacks until it is sure that the configuration is compatible with an initialization by i. At the end of $\mathcal{RI}_i^{\mathcal{A}}$, i can initialize $\mathcal{A}$.

PROTOCOL $\mathcal{L}_i^{\mathcal{A}}$
/* $\mathcal{L}_i^{\mathcal{A}}$ is called either in Line 2 of $\mathcal{H}_i^{\mathcal{A}}$ or upon the receipt of some broadcasts from $\mathcal{RI}^{\mathcal{A}}$. */
/* In case of several broadcasts, they are all handled simultaneously. */
/* However, after that, no new broadcast from $\mathcal{RI}^{\mathcal{A}}$ is accepted at i * until $\mathcal{L}_i^{\mathcal{A}}$ terminates. */
/* Moreover, all feedbacks from $\mathcal{RI}^{\mathcal{A}}$ are locked at i until $\mathcal{L}_i^{\mathcal{A}}$ terminates. */
/* Once $\mathcal{L}_i^{\mathcal{A}}$ is terminated, any enabled feedback is executed before i * starts $\mathcal{L}_i^{\mathcal{A}}$ again. */
(1) IF $\mathcal{LE}_i$ RETURNS i THEN
(2) WHILE ($\mathcal{LE}_i$ RETURNS i) AND ($\mathcal{TD}_i^{\mathcal{CR}^{\mathcal{A}}}$ RETURNS $\neg\texttt{TD}$) DO;
(3) $\mathcal{CR}_i^{\mathcal{A}}$
(4) ENDIF

For every process j, if $j \neq \ell$, the execution of $\mathcal{L}_j^{\mathcal{A}}$ is simply execution of $\mathcal{LE}_j$. Otherwise, the leader ℓ should execute another algorithm, called Algorithm $\mathcal{CR}^{\mathcal{A}}$ (see Line 3). However, before initiating $\mathcal{CR}^{\mathcal{A}}$, ℓ must ensure that no other algorithm $\mathcal{CR}^{\mathcal{A}}$ is running in the system. So, we need a snap-stabilizing termination detector rooted at ℓ ($\mathcal{TD}^{\mathcal{CR}^{\mathcal{A}}}$), which computes the predicate $\texttt{TD}$ defined as follows: $\texttt{TD}$ is true if and only if there exists no algorithm $\mathcal{CR}^{\mathcal{A}}$ running on the system. Once ℓ exits the "while loop" in Line 2 of $\mathcal{L}_\ell^{\mathcal{A}}$, it is guaranteed to execute $\mathcal{CR}^{\mathcal{A}}$ alone.

Similar to the case of a unique initiator, we use a snap-stabilizing snapshot algorithm $\mathcal{S}^{\mathcal{A}}$ to design $\mathcal{CR}^{\mathcal{A}}$. Algorithm $\mathcal{S}^{\mathcal{A}}$ computes the predicate $\texttt{OK}$ related to $\mathcal{A}$ (Line 2). Then, if the result of the snapshot is $\neg\texttt{OK}$, Reset $\mathcal{R}^{\mathcal{A}}$ is executed (Line 3).

PROTOCOL $\mathcal{CR}_i^{\mathcal{A}}$
(1) IF $\mathcal{LE}_i$ RETURNS i THEN /* i is the actual leader */
(2) $\mathcal{S}_i^{\mathcal{A}}$
(3) IF $\neg\texttt{OK}$ THEN $\mathcal{R}_i^{\mathcal{A}}$ ENDIF
(4) ENDIF

So, at the termination of $\mathcal{H}_i^{\mathcal{A}}$, the system is in a normal configuration and i can initialize $\mathcal{A}$.

In summary, an initiator i executes the algorithm $\mathcal{H}_i^{\mathcal{A}} \rightarrow \mathcal{A}$. Upon receiving a broadcast of some $\mathcal{RI}_i^{\mathcal{A}}$ with $i \neq j$, a non-initiator j executes $\mathcal{L}_j^{\mathcal{A}}$ in parallel with $\mathcal{A}$. If j is not involved into any $\mathcal{RI}^{\mathcal{A}}$, then j either continues executing $\mathcal{A}$ if $\mathcal{A}$ is running nor does anything if there is no execution of $\mathcal{A}$.

However due to the arbitrary initial configuration, an execution of $\mathcal{A}$ which has not been properly initiated can lead the system to a deadlock or a livelock. So, to prevent such a situation we use a control algorithm: $\mathcal{PCR}_j^{\mathcal{A}}$ controls of the correctness of the configurations. As previously, this control has to be executed by the leader alone. But, it has to be implemented at each process. Thus, the first action is to check if the process is the actual leader. In that case, the leader executes $\mathcal{CR}^{\mathcal{A}}$ (line 2) to check the correctness of the configuration, and performs a reset if the system is in an abnormal configuration.

PROTOCOL $\mathcal{PCR}_i^{\mathcal{A}}$
(1) IF $\mathcal{LE}_i$ RETURNS i THEN /* i is the actual leader */
(2) $\mathcal{CR}_i^{\mathcal{A}}$
(3) ENDIF

Hence, the transformed version, $\texttt{TRANS2}_{\mathcal{A}}$, of $\mathcal{A}$ is the fair parallel composition $\mathcal{A}||\mathcal{SNAP}$, where $\mathcal{SNAP}$ is defined in Figure 6.2.

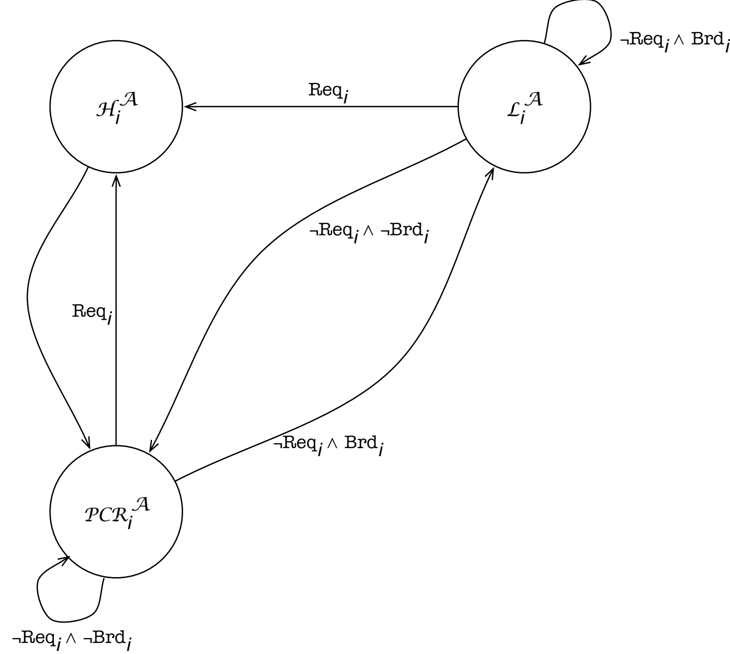

Figure 6.2: $\mathcal{SNAP}_i$

Remark 6.3 *$\mathcal{LE}$, $\mathcal{S}^{\mathcal{A}}$, $\mathcal{TD}$, and $\mathcal{R}^{\mathcal{A}}$ are snap-stabilizing finite algorithms (since they are based on the snap-stabilizing algorithm $\mathcal{PIF}$).*

From Remark 6.3, we obtain the following result:

Lemma 6.1 *Every execution of Algorithm $\mathcal{CR}^{\mathcal{A}}$ is finite (even without any starting action).*

By Remark 6.3 and Lemma 6.1, we can claim the following:

Corollary 6.1 *Every execution of Algorithm $\mathcal{PCR}^{\mathcal{A}}$ is finite (even without any starting action).*

Lemma 6.2 *Every process which is not the actual leader of the network can execute Algorithm $\mathcal{CR}^{\mathcal{A}}$ at most once.*

Proof. Any process i executes $\mathcal{CR}^{\mathcal{A}}$ inside $\mathcal{L}^{\mathcal{A}}$ and $\mathcal{PCR}^{\mathcal{A}}$. Since they are never executed in parallel, to execute $\mathcal{CR}^{\mathcal{A}}$ twice, a process i has to initiate $\mathcal{LE}$. As the result of $\mathcal{LE}$ is different from i, and consequently i cannot execute $\mathcal{CR}^{\mathcal{A}}$. □

Lemma 6.3 *Every execution of Algorithm $\mathcal{L}^{\mathcal{A}}$ is finite (even without any starting action).*

Proof. First from Remark 6.3, Line 1 terminates. Consider then the execution of the while loop (Line 2 of $\mathcal{L}^{\mathcal{A}}$). If the process is not the leader, the loop cannot be executed more than twice (due to the result of $\mathcal{LE}$). Otherwise,

by Lemma 6.2, overall the processes which are not the leader will execute $\mathcal{CR}^{\mathcal{A}}$ at most $n-1$ times. So, eventually, $\mathcal{TD}^{\mathcal{CR}^{\mathcal{A}}}$ will return TD equal to true, and Line 2 also terminates. Finally, by Lemma 6.1, Line 3 terminates. □

$\mathcal{RI}_i^{\mathcal{A}}$ is a snap-stabilizing $\mathcal{PIF}$ where the broadcast and feedback phases can only be locally blocked on some process j because it is executing $\mathcal{L}_j^{\mathcal{A}}$. Now, once $\mathcal{L}_j^{\mathcal{A}}$ is terminated, j executes its enabled feedbacks of $\mathcal{RI}^{\mathcal{A}}$ before starting $\mathcal{L}_j^{\mathcal{A}}$ again. Hence, by Lemma 6.3, we can conclude with the following corollary.

Corollary 6.2 *Every execution of Algorithm $\mathcal{RI}^{\mathcal{A}}$ is finite (even without any starting action).*

By Remark 6.3 and Corollary 6.2, we can state the following:

Corollary 6.3 *Every execution of Algorithm $\mathcal{H}^{\mathcal{A}}$ is finite (even without any starting action).*

Lemma 6.4 *At the end of the first execution of Algorithm $\mathcal{L}^{\mathcal{A}}$ initiated by the leader, the system is in a normal configuration (w.r.t. $\mathcal{A}$) and no more resets will be initiated.*

Proof. Since the execution of $\mathcal{L}^{\mathcal{A}}$ is finite (Lemma 6.3), during the last execution of $\mathcal{TD}^{\mathcal{CR}^{\mathcal{A}}}$, no process was running $\mathcal{CR}^{\mathcal{A}}$ while executing the part of $\mathcal{TD}^{\mathcal{CR}^{\mathcal{A}}}$. So, if a process later executes $\mathcal{CR}^{\mathcal{A}}$, it must have initiated the execution. Since it is not the leader, it cannot execute anything other than the leader election part ($\mathcal{LE}$). So, at the end of the while loop, no process except the leader can execute a snapshot or a reset. When the leader ends $\mathcal{CR}^{\mathcal{A}}$, either the system has been reset to a normal starting configuration (for $\mathcal{A}$), or the system is in a normal configuration (not necessarily a starting configuration) because the result of $\mathcal{S}^{\mathcal{A}}$ was a normal configuration (OK was true). □

Corollary 6.4 *$\mathcal{L}^{\mathcal{A}}$ is a snap-stabilizing finite algorithm for the following specification: if i is the leader, then at the end of $\mathcal{L}_i^{\mathcal{A}}$, the system is normal (w.r.t. $\mathcal{A}$) and no more resets will be initiated.*

Lemma 6.5 *The leader executes $\mathcal{CR}^{\mathcal{A}}$ infinitely many times.*

Proof. Assume that the leader ℓ only executes $\mathcal{L}_\ell^{\mathcal{A}}$ and $\mathcal{H}_\ell^{\mathcal{A}}$ finitely many times. Then ℓ executes $\mathcal{PCR}^{\mathcal{A}}$ infinitely many times. $\mathcal{LE}_\ell$ will return ℓ infinitely many times, so ℓ executes $\mathcal{CR}^{\mathcal{A}}$ infinitely many times. □

Lemma 6.6 *Within finite time, the system is in a normal configuration* w.r.t. $\mathcal{A}$.

Proof. By Lemmas 6.1, 6.2, and 6.5, eventually the leader executes $\mathcal{CR}^{\mathcal{A}}$ alone. Consider the first time $\mathcal{CR}^{\mathcal{A}}$ is started by the leader alone. When that execution terminates, either the system has been reset to a normal starting configuration (for $\mathcal{A}$), or the system is in a normal configuration (not necessarily a starting configuration) because the result of $\mathcal{S}^{\mathcal{A}}$ was a normal configuration (OK was true). □

Theorem 6.2 TRANS2$_{\mathcal{A}}$ *is snap-stabilizing under the distributed weakly fair daemon w.r.t. the specification of the problem solved by $\mathcal{A}$.*

Proof. Let i be an initiator of $\mathcal{A}$. By Lemma 6.3, and Corollaries 6.1 and 6.3, i eventually launches $\mathcal{H}_i^{\mathcal{A}}$. By Corollary 6.3, $\mathcal{H}_i^{\mathcal{A}}$ is performed in finite time. Now, during this execution of $\mathcal{H}_i^{\mathcal{A}}$, the leader ℓ must have executed $\mathcal{L}_\ell^{\mathcal{A}}$. So, once $\mathcal{H}_i^{\mathcal{A}}$ is done, the system is in a normal configuration *w.r.t.* $\mathcal{A}$ and will not be reset any more, by Lemma 6.4. Hence, from that point onwards, i will eventually start $\mathcal{A}$ (by Remark 6.2) and $\mathcal{A}$ will behave as expected (in particular, from the previous lemmas and corollaries, $\mathcal{A}$ will never be deadlocked). □

Note that the implementation of TRANS2$_{\mathcal{A}}$ in the locally shared memory model follows the same guidelines as the ones discussed at the end of Section 6.1 for TRANS1$_{\mathcal{A}}$.

Let us now consider the issue of fairness. We borrow the same ideas as in [37, 38]. We first construct an algorithm called $\mathcal{TF}$ which is *total fair* [38] in any identified network, meaning that each of its executions under the distributed unfair daemon contains an infinity of actions at each process. Then, we compose TRANS2$_{\mathcal{A}}$ with $\mathcal{TF}$ using the *cross-over composition* [37]—details are given below. The resulting algorithm, noted TRANS2$_{\mathcal{A}} \diamond \mathcal{TF}$, satisfies the same safety properties as TRANS2$_{\mathcal{A}}$. Moreover, it enforces the liveness properties of TRANS2$_{\mathcal{A}}$ in such a way that they now

endure a distributed unfair daemon. Hence, we obtain our final result: $\texttt{TRANS2}_{\mathcal{A}} \diamond \mathcal{TF}$ is snap-stabilizing under the distributed unfair daemon *w.r.t.* the specification of the problem solved by $\mathcal{A}$.

First, to construct the total fair algorithm $\mathcal{TF}$ we proceed as follows. We endow each process i with one instance of Algorithm $\mathcal{PIF}$, noted $\mathcal{PIF}_i$. Each process i executes infinitely many $\mathcal{PIF}_i$ waves in sequence. Notice that for every i, j such that $i \neq j$, instances of $\mathcal{PIF}_i$ and $\mathcal{PIF}_j$ run in parallel. Now, since

- the number of processes is finite,
- there are exactly as many instances of $\mathcal{PIF}$ as processes which run at a time (each instance $\mathcal{PIF}_i$ executes its own waves in sequence), and
- each complete wave of each instance $\mathcal{PIF}_i$ is executed within a finite number of steps (Corollary 4.4, page 23),

we can deduce that at least one instance $\mathcal{PIF}_i$ performs infinitely many complete $\mathcal{PIF}$ waves in any execution assuming a distributed unfair daemon. By definition, each complete $\mathcal{PIF}$ wave involves all processes. Consequently, $\mathcal{TF}$ is a total fair algorithm.

Then, by composing our transformer $\texttt{TRANS2}_{\mathcal{A}}$ (which works assuming a distributed weakly fair daemon, see Theorem 6.2) with $\mathcal{TF}$ using the cross-over composition, we obtain a transformer which works assuming a distributed unfair daemon. Indeed, the *cross-over* composition has been designed as a tool for scheduler management. Informally, in the composition $A \diamond B$, the (initial) actions of A are synchronized with actions of B: the actions of A are performed only when an action of B is performed too, in the same step. So, the execution of A is fully driven by B, *i.e.*, B acts as a scheduler for A. Hence, if B is a total fair algorithm, then in any execution of $A \diamond B$, every process which is continuously enabled *w.r.t.* A executes an action of A within finite time, and we are done.

Formally, the cross-over composition $A \diamond B$ consists in the following rewriting rules. For every action $L_A :: G_A \to S_A$ of A, and for every action $L_B :: G_B \to S_B$ of B, Algorithm $A \diamond B$ contains the action

$$L_{A,B} :: G_A \wedge G_B \to S_A; S_B$$

Moreover, for every action $L_B :: G_B \to S_B$ of B, Algorithm $A \diamond B$ contains the action

$$L_{onlyB} :: G_B \wedge \neg G_{all} \to S_B$$

where G_{all} is the disjunction of all guards of actions in A.

Hence, with $\texttt{TRANS2}_{\mathcal{A}} \diamond \mathcal{TF}$, we obtain the following result.

Theorem 6.3 *Let $\mathcal{A}$ be any multi-initiator distributed algorithm. Then, $\mathcal{A}$ can be transformed into a snap-stabilizing algorithm for the same specification working under a distributed unfair daemon.*

From Theorems 6.1 and 6.3, we can claim the following final result:

Theorem 6.4 *Let $\mathcal{A}$ be any single- or multi-initiator algorithm satisfying Remark 6.1. Then, $\mathcal{A}$ can be transformed into a snap-stabilizing algorithm for the same specification working under a distributed unfair daemon.*

7 Conclusion

In this paper, we assumed the same hypothesis as in [28], *i.e.*, we consider any distributed algorithm that can be semi-automatically (modulo a predicate on configurations) transformed into a self-stabilizing algorithm by the transformer of [28]. Yet, while message-passing is assumed in [28], we consider here a stronger model, the local shared memory model of communication. Our goal was to study the expressiveness of snap-stabilization in that model.

To achieve this, we first explained how to automatically generate dynamic specifications of problems so that we can design snap-stabilizing algorithms. We then presented the first snap-stabilizing Propagation of Information with Feedback (PIF) algorithm for asynchronous arbitrary networks which is proven assuming a distributed unfair daemon. This algorithm is a key module in the sense that it can be used to design numerous snap-stabilizing algorithms, *e.g.*, reset, snapshot, leader election, and termination detection. We demonstrated that this PIF algorithm is also a critical tool to transform almost any algorithm into a snap-stabilizing version, semi-automatically. Our PIF-based transformer

shows that, in the locally shared memory model, self-stabilization and snap-stabilization have the same expressiveness, *i.e.*, any problem that admits a self-stabilizing algorithm also admits a snap-stabilizing solution, and *vice versa*.

Due to the great safety feature of snap-stabilization, it is now crucial to solve this open question : "*What is the expressiveness of snap-stabilization in asynchronous message passing?*" Following the impossibility result of [26], this question cannot hold for deterministic snap-stabilizing algorithms on systems with unbounded or unknown capacity channels. Nevertheless, the expressiveness of snap-stabilization in the asynchronous message-passing model with known bounds on link capacity is a challenging open question, in particular when considering identified networks of arbitrary topologies.[3]

References

[1] A Cournier, AK Datta, F Petit, and V Villain. Enabling snap-stabilization. In *23th International Conference on Distributed Computing Systems (ICDCS 2003)*, pages 12–19, Providence, Rhode Island USA, May 19-22 2003. IEEE Computer Society Press.

[2] A Cournier, S Devismes, and V Villain. Snap-stabilizing PIF and useless computations. In *The Twelfth International Conference on Parallel and Distributed Systems (ICPADS'06)*, volume 1, pages 39–46, Minneapolis, USA, 2006. IEEE Computer Society Press P2612.

[3] EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.

[4] Sukumar Ghosh, Arobinda Gupta, Ted Herman, and Sriram V. Pemmaraju. Fault-containing self-stabilizing distributed protocols. *Distributed Computing*, 20(1):53–73, 2007.

[5] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 1997, 1997.

[6] A Bui, AK Datta, F Petit, and V Villain. State-optimal snap-stabilizing PIF in tree networks. In *Proceedings of the Fourth Workshop on Self-Stabilizing Systems*, pages 78–85, Austin, Texas, USA, June 1999. IEEE Computer Society Press.

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

[8] A Bui, AK Datta, F Petit, and V Villain. Snap-stabilizing PIF algorithm in tree networks without sense of direction. In *SIROCCO'99, The 6th International Colloquium On Structural Information and Communication Complexity Proceedings*, pages 32–46. Carleton University Press, 1999.

[9] Alain Cournier, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. Optimal snap-stabilizing PIF algorithms in un-oriented trees. *J. High Speed Networks*, 14(2):185–200, 2005.

[10] A Cournier, AK Datta, F Petit, and V Villain. Snap-stabilizing PIF algorithm in arbitrary rooted networks. In *22st International Conference on Distributed Computing Systems (ICDCS-22)*, pages 199–206. IEEE Computer Society Press, 2002.

[11] L Blin, A Cournier, and V Villain. An improved snap-stabilizing PIF algorithm. In *DSN SSS'03 Workshop: Sixth Symposium on Self-Stabilizing Systems (SSS'03)*, pages 199–214. LNCS 2704, 2003.

[12] Alain Cournier, Stéphane Devismes, and Vincent Villain. Light enabling snap-stabilization of fundamental protocols. *TAAS*, 4(1), 2009.

[13] Franck Petit and Vincent Villain. Optimal snap-stabilizing depth-first token circulation in tree networks. *J. Parallel Distrib. Comput.*, 67(1):1–12, 2007.

[3] Notice that snap-stabilizing algorithms for some dedicated problems in particular topologies already exist in the asynchronous message-passing model with known bounds on link capacity, *e.g.*, [25, 26, 27].

[14] A Cournier, S Devismes, and V Villain. A snap-stabilizing DFS with a lower space requirement. In *Seventh International Symposium on Self-Stabilizing Systems (SSS'05)*, pages 33–47, Barcelona, Spain, 2005. LNCS 3764.

[15] A Cournier, S Devismes, F Petit, and V Villain. Snap-Stabilizing Depth-First Search on Arbitrary Networks. *The Computer Journal*, 49(3):268–280, 2006.

[16] D Bein, AK Datta, and V Villain. Snap-stabilizing optimal binary search tree. In *Seventh International Symposium on Self-Stabilizing Systems (SSS'05)*, pages 1–17, Barcelona, Spain, 2005. LNCS 3764.

[17] A Cournier, S Devismes, and V Villain. Snap-stabilizing detection of cutsets. In *HIPC 2005, 12th Annual IEEE Conference on High Performance Computing*, pages 488–497. LNCS 3769, 2005.

[18] Colette Johnen, Luc Alima, Ajoy K. Datta, and Sébastien Tixeuil. Optimal snap-stabilizing neighborhood synchronizer in tree networks. *Parallel Processing Letters*, 12(3-4):327–340, 2002.

[19] Florent Nolot. *Stabilisation des horloges de phases dans les systèmes distribués*. PhD thesis, Université de Picardie Jules Verne, LaRIA, Amiens, 2002.

[20] Borzoo Bonakdarpour, Stéphane Devismes, and Franck Petit. Snap-stabilizing committee coordination. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May*, pages 231–242. IEEE, 2011.

[21] Eddy Caron, Frédéric Desprez, Franck Petit, and Cédric Tedeschi. Snap-stabilizing prefix tree for peer-to-peer systems. *Parallel Processing Letters*, 20(1):15–30, 2010.

[22] Alain Cournier, Swan Dubois, and Vincent Villain. A snap-stabilizing point-to-point communication protocol in message-switched networks. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, pages 1–11. IEEE, 2009.

[23] Alain Cournier, Swan Dubois, Anissa Lamani, Franck Petit, and Vincent Villain. Snap-stabilizing linear message forwarding. In Shlomi Dolev, Jorge Arturo Cobb, Michael J. Fischer, and Moti Yung, editors, *Stabilization, Safety, and Security of Distributed Systems - 12th International Symposium, SSS 2010, New York, NY, USA, September 20-22*, volume 6366 of *Lecture Notes in Computer Science*, pages 546–559. Springer, 2010.

[24] Alain Cournier, Swan Dubois, Anissa Lamani, Franck Petit, and Vincent Villain. The snap-stabilizing message forwarding algorithm on tree topologies. *Theor. Comput. Sci.*, 496:89–112, 2013.

[25] Shlomi Dolev and Nir Tzachar. Empire of colonies: Self-stabilizing and self-organizing distributed algorithm. *Theor. Comput. Sci.*, 410(6-7):514–532, 2009.

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

[27] Khaled Mohamed, Florence Levé, and Vincent Villain. Snap-stabilizing PIF on non-oriented trees and message passing model. In *16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 299–313, Padeborn, Germany, Sept 28 - Oct 1 2014.

[28] S Katz and KJ Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7:17–26, 1993.

[29] B Awerbuch, S Kutten, Y Mansour, B Patt-Shamir, and G Varghese. Time optimal self-stabilizing synchronization. In *STOC93 Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 652–661, 1993.

[30] N Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

[31] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

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

[33] EJH Chang. Echo algorithms: depth parallel operations on general graphs. *IEEE Transactions on Software Engineering*, SE-8:391–401, 1982.

[34] A Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, IT-29:23–35, 1983.

[35] B Awerbuch and R Ostrovsky. Memory-efficient and self-stabilizing network reset. In *PODC94 Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 254–263, 1994.

[36] KM Chandy and L Lamport. Distributed snapshots: determining global states od distributed systems. *ACM Transactions on Computers Systems*, 3(1):63–75, 1985.

[37] Joffroy Beauquier, Maria Gradinariu, and Colette Johnen. Cross-over composition - enforcement of fairness under unfair adversary. In *Self-Stabilizing Systems, 5th International Workshop, WSS 2001, Lisbon, Portugal, October 1-2, 2001, Proceedings*, volume 2194 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2001.

[38] Joffroy Beauquier, Maria Gradinariu, and Colette Johnen. Randomized self-stabilizing and space optimal leader election under arbitrary scheduler on rings. *Distributed Computing*, 20(1):75–93, 2007.