Light Enabling Snap-Stabilization of Fundamental Protocols

ALAIN COURNIER, STEPHANE DEVISMES, and VINCENT VILLAIN
LaRIA CNRS

In this article, we show that some fundamental self- and snap-stabilizing wave protocols (e.g., token circulation, *PIF*, etc.) implicitly assume a very light property that we call *BreakingIn*. We prove that *BreakingIn* is strictly induced by self- and snap-stabilization. Combined with a transformer, *BreakingIn* allows to easily turn the non-fault-tolerant versions of those protocols into snap-stabilizing versions. Unlike the previous solutions, the transformed protocols are very efficient and work at least with the same daemon as the initial versions extended to satisfy *BreakingIn*. Finally, we show how to use an additional property of the transformer to design snap-stabilizing extensions of those fundamental protocols like Mutual Exclusion.

Categories and Subject Descriptors: C.2.4 [**Distributed Systems**]: Distributed applications

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Self- and snap-stabilization, transformer, wave protocols

ACM Reference Format:

Cournier, A., Devismes, S., and Villain, V. 2009. Light enabling snap-stabilization of fundamental protocols. ACM Trans. Autonom. Adapt. Syst. 4, 1, Article 6 (January 2009), 27 pages. DOI = 10.1145/1462187.1462193 http://doi.acm.org/10.1145/1462187.1462193

1. INTRODUCTION

Stabilization is a nice approach to design distributed systems tolerating transient faults. This notion first appears with the *self-stabilization* defined in Dijkstra [1974]: A self-stabilizing system, regardless of its initial state, is guaranteed to converge into the intended behavior in finite time. *Snap-stabilization* is introduced in Bui et al. [1999]: Starting from any configuration, a *snap-stabilizing protocol* always behaves according to the specifications of the problem to be solved. In other words, a snap-stabilizing protocol is a self-stabilizing protocol which stabilizes in 0 time unit. It is important to note that the zero

This article is a revised shortened version of Cournier et al. [2007].

Authors' address: LaRIA CNRS FRE 2733, Université de Picardie Jules Verne, Amiens, France; email: Stephane.devismes@u-picardie.fr.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2009 ACM 1556-4665/2009/1-ART6 $5.00 DOI 10.1145/1462187.1462193 http://doi.acm.org/10.1145/1462187.1462193

stabilization time does not guarantee that all components of the system never work in a fuzzy manner. The snap-stabilization actually ensures two properties despite the arbitrary initial configuration: (1) The protocol can start in finite time using special actions called *starting actions* and such actions are triggered by external (with regard to the protocol) requests; (2) since a starting action is executed, the protocol behaves as expected. With such properties, the system always satisfies its specifications. Indeed, when the protocol receives a request, this means that an external application (or a user) requests the computation of a specific task provided by the protocol. In this case, a snap-stabilizing protocol guarantees that the requested task is executed as expected. On the contrary, while the protocol receives no request, it has nothing to guarantee. Consider, for instance, the problem of *mutual exclusion*. Starting from any configuration, a snap-stabilizing protocol cannot prevent several (nonrequesting) processes from executing the critical section simultaneously. However, it guarantees that every requesting process executes the critical section alone. Usually, a self-stabilizing protocol does not provide such a guarantee: While a snap-stabilizing protocol ensures that any request is satisfied despite the arbitrary initial configuration, a self-stabilizing protocol often needs to be repeated an unbounded number of times before guaranteeing the satisfaction of any request.

Some protocols, called *transformers*, were proposed to automatically stabilize protocols [Katz and Perry 1993; Cournier et al. 2003]. In Katz and Perry [1993] use self-stabilizing snapshots to design a protocol that transforms almost all non-self-stabilizing protocols designed for message-passing model into self-stabilizing protocols working in the same model. A *transformer* based on a snap-stabilizing *Propagation of Information with Feedback* (*PIF*) is proposed in Cournier et al. [2003]. This transformer provides a snap-stabilizing version of any protocol that can be self-stabilized by the transformer of Katz and Perry [1993]. However, this transformer is designed in a higher level model than the previous one (n.b., there are some snap-stabilizing protocols working in the message-passing model [Dolev and Tzachar 2006]). The goal of these transformers [Katz and Perry 1993; Cournier et al. 2003] is not to efficiently transform protocols. Actually, they are used to demonstrate the possibility of stabilizing a wide class of protocols. These transformers use heavy mechanisms to transform an initial protocol into a stabilizing protocol, and the overhead of the stabilization is often difficult to evaluate. In particular, they use snapshots to regularly evaluate a predicate defined on the variables of the protocol to transform. This predicate characterizes the normal configurations of the system and requires IDs on processes. This technique allows one to prevent the system from deadlocks and livelocks. The main drawbacks of this technique are: (1) Such a predicate is generally difficult to formalize and (2) the number of snapshots used by the transformer cannot be bounded compared to the number of actions executed by the initial protocol to transform. This latter drawback results in two important consequences. First, the overhead of the transformation cannot be bounded. Second, the transformed protocol requires some fairness assumptions.

In this paper we propose a light semi-automatic method to efficiently transform non fault-tolerant protocols into snap-stabilizing protocols.[1] We focus on single-initiator wave protocols where decisions only occur at the initiator. Although this protocol class seems to be restrictive, it gathers some fundamental protocols like token circulation, *PIF*, etc. Also, these protocols are key tools to solve a wide class of problems: *Reset*, *Snapshot*, *Mutual Exclusion*, *Routing*, *Synchronisation*, etc. Our approach is the following: We first show that any (self- or snap-) stabilizing version of such protocols implicitly assume a light property which we call *BreakingIn*. We prove that *BreakingIn* is strictly induced by self- and snap-stabilization. In other words, *BreakingIn* is easier to obtain than self- or snap-stabilization. We then propose a way to slightly update a non-fault-tolerant protocol $\mathcal{A}$ into a nonstabilizing protocol $\mathcal{B}$ satisfying *BreakingIn*. Combining $\mathcal{B}$ with a transformer, we obtain a snap-stabilizing version of $\mathcal{A}$. The main advantage of our method is that our transformer does not use snapshots. In consequence and in contrast with Cournier et al. [2003]: (1) We do not use a predicate that characterizes the normal configuration of the system (in consequence, we do not require IDs on processes); (2) the overhead of the snap-stabilization is now appraisable; and (3) the transformed protocols work at least under the same daemon as the initial versions extended to satisfy *BreakingIn* (this latter result allows us to design snap-stabilizing protocols working under an unfair daemon). To illustrate the power of our method, we then propose two snap-stabilizing applications designed with our transformer: A depth-first token circulation and a breath-first spanning tree construction. The complexity analysis of these two protocols will show their efficiency in both time and space. Also, the simplicity of their codes will show that in practice as in theory *BreakingIn* is easier to obtain than stabilization. Hence, in contrast with the specific (self- or snap-) stabilizing solutions that usually implement complex mechanism [Huang and Chen 1993; Johnen 1997; Cournier et al. 2005; Cournier et al. 2006], our transformer can be seen as a powerful tool to simplify the design of snap-stabilizing protocols. Finally, we will show that, thanks to a counting property of our transformer, our depth-first token circulation can be used to solve in few lines the *mutual exclusion* in a snap-stabilizing manner.

The rest of the article is organized as follows: We describe our model in Section 2. In Section 3, we present a method to semi-automatically snap-stabilize protocols. We propose two applications designed with our method in Section 4. Using the depth-first token circulation proposed in Section 4, we design a snap-stabilizing mutual exclusion in Section 5. Finally, we conclude in Section 6.

2. PRELIMINARIES

2.1 Network

We consider a *network* as any undirected connected graph $G = (V,E)$ where V is the set of *processes* and E is the set of *bidirectional links*. The network is assumed to be *rooted*, that is, we distinguish a special process called *root* and

[1]This approach is a generalization of the method we propose in Cournier et al. [2006a] to transform some self-stabilizing protocols into snap-stabilizing ones.

noted r. The root corresponds to the *protocol initiator*. Two processes p and q are said *neighbors iff* the link (p,q) exists. Every process can distinguish all its incident links. For sake of simplicity, we refer to the link (p,q) by the *label* q in the code of p. We assume that the labels of p, stored in $Neig_p$, are locally ordered by any arbitrary order relation noted $\prec_p$. We also use the following notations: N is the size of the network, Δ its degree, and D its diameter.

2.2 Computational Model

We consider a locally shared memory model where the program of every process consists of a finite set of *shared variables* and a finite set of *actions*. Each process can write into its own variables only and read its own variables and that of its neighbors. Each action is constitued as follows: $\langle label \rangle :: \langle guard \rangle \rightarrow \langle statement \rangle$. The guard of an action of p is a boolean expression involving variables of p and its neighbors. The statement of an action of p updates some variables of p. An action is executed only if its guard is satisfied.

The *state* of a process is defined by the value of its variables. The *configuration* of a system is the product of the states of all processes. We note $\mathcal{C}$ the set of all configurations of the system. Let $\gamma \in \mathcal{C}$ and A an action of $p \in V$. A is said *enabled* at p in γ *iff* the guard of A is satisfied by p in γ. p is said to be *enabled* in γ *iff* at least one action is enabled at p in γ.

To simplify the design of the programs, we assume priorities on the actions. The priorities follow the order of appearance of the actions into the text of the programs. When several actions of a process are simultaneously enabled, only its priority enabled action can be activated. Let a distributed protocol $\mathcal{P}$ be a collection of binary transition relations on $\mathcal{C}$, noted $\mapsto$. An *execution* of $\mathcal{P}$ is a *maximal* sequence of configurations $e = \gamma_0,\gamma_1,\ldots,\gamma_i,\gamma_{i+1},\ldots$ such that, $\forall i \geq 0$, $\gamma_i \mapsto \gamma_{i+1}$ (called a *step*) if γ_{i+1} exists, else γ_i is a terminal configuration. Each step $\gamma_i \mapsto \gamma_{i+1}$ is shared into three phases atomically executed: (1) Every process evaluates its guards, (2) a *daemon* chooses some enabled process, and (3) each chosen process executes its priority enabled action. When the three phases are done, the next step begins. A *daemon* is defined in terms of *fairness* and *distribution*. There exist several kinds of fairness assumptions. Here, we consider the *strongly fairness*, *weakly fairness*, and *unfairness* assumptions. Under a *strongly fair* daemon, every process that is enabled infinitely often is eventually chosen by the daemon. When a daemon is *weakly fair*, every continuously enabled process is eventually chosen by the daemon. Finally, the *unfair* daemon is the weakest scheduling assumption: It can forever prevent a process to execute an action except if 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 of these processes to execute an action. To simplify the notation, we denote (when necessary) the distributed *strongly fair*, *weakly fair*, and *unfair* daemons by *SF*, *WF*, and *UF*, respectively.

We consider that a process p is *neutralized* in $\gamma_i \mapsto \gamma_{i+1}$ if p was *enabled* in γ_i and not enabled in γ_{i+1}, but did not execute any action in $\gamma_i \mapsto \gamma_{i+1}$.

To compute the time complexity, we use the notion of *round* [Dolev et al. 1997]. This notion captures the execution rate of the slowest process in any execution. The first *round* of an execution e, noted e', is the minimal prefix of e containing the execution of one action or the neutralization of every enabled process from the initial configuration. Let e'' be the suffix of e such that $e = e'e''$. The second *round* of e is the first round of e'', and so on.

2.3 Wave Protocols

The wave protocols are defined in Tel [2001] as follows:

Definition 2.1 (*Wave Protocol*). A *wave protocol* is a protocol $\mathcal{P}$ satisfying: (1) Each execution of $\mathcal{P}$ is finite and contains at least one *decision*. (2) Each decision is causally preceded by an action of each process.

An execution of a wave protocols is called *wave*. In each wave, there is a special type of actions called *decision*. Roughly speaking, this action occurs when the task provided by the protocol is done. For instance, in a token circulation, the decision occurs only after the token visits all the processes. As an additional notation, in any execution a distinction is made between the *initiators* and the *noninitiators*. An initiator starts the execution of its local algorithm without being causally preceded by any other internal (with regard to the protocol) action of any process, that is, the start is triggered by an external action called *request*. Such a request is assumed to be executed by an application or a user which needs an execution of the protocol. The notion of *request* is usually implicit in distributed systems: The starting actions are assumed to be executed spontaneously. In particular, in self-stabilizing wave protocols, the waves need to be repeated to guarantee the convergence to a specified behavior and, as a consequence, the notion of request is simply kept in the background. On the contrary, a snap-stabilizing protocol, despite its arbitrary initial configuration, works according to its specifications after its first execution. Hence, in such protocols, the request becomes primordial again. In this paper, although we do not explicitly express the request into the code of our protocol, we assume that any initiator can executed a starting action *only if*: (1) This action is effectively enabled and (2) an application (or a user) requested it. In the appendix (Subsection 6), we show how to explicitly express the request in a protocol.

2.4 Snap-Stabilization

Let $\mathcal{T}$ be a task, $\mathcal{F}$ a specification of $\mathcal{T}$, and $\mathcal{P}$ a protocol.

Definition 2.2 (*Snap-Stabilization*). $\mathcal{P}$ is snap-stabilizing for $\mathcal{F}$ *iff* starting from any configuration, every execution of $\mathcal{P}$ satisfies $\mathcal{F}$.

If we consider now that $\mathcal{P}$ is a wave protocol, we can clarify the definition as follows:

Remark 2.3. Any wave protocol $\mathcal{P}$ is snap-stabilizing for $\mathcal{F}$ *iff*: (1) $\mathcal{P}$ starts any requested task $\mathcal{T}$ in finite time, and (2) from any configuration where $\mathcal{P}$ starts, it computes $\mathcal{T}$ according to $\mathcal{F}$.

3. THE TRANSFORMER

3.1 Principle

Let $\mathcal{A}$ be a non-fault-tolerant single-initiator wave protocol where decisions only occur at the initiator. Let $\mathcal{T}$ be the task solved by $\mathcal{A}$. We want to efficiently snap-stabilize $\mathcal{A}$ in both space and time complexity. As a consequence, we have to snap-stabilize $\mathcal{A}$ without using snapshots. In Katz and Perry [1993] and Cournier et al. [2003], the snapshots are used for detecting deadlocks and livelocks in the execution of the initial protocol. As we do not want to use snapshots, we cannot detect the deadlocks or livelocks that may occur when using $\mathcal{A}$ starting from an arbitrary configuration. So, we propose to slightly modify $\mathcal{A}$ to obtain a protocol $\mathcal{B}$ satisfying an additional property. This property must allow $\mathcal{B}$ to be automatically snap-stabilized by a transformer protocol that we call *Black Box*. Also, this property must be as simple as possible. In particular, it must be easier to design than self- or snap-stabilization. Let $\mathcal{SSBB}(\mathcal{B})$ be the snap-stabilizing version of $\mathcal{B}$ obtained with our Snap-Stabilizing Black Box ($\mathcal{SSBB}$). By Remark 2.3, $\mathcal{SSBB}(\mathcal{B})$ must satisfy two properties starting from any configuration:

—Upon an external request, $\mathcal{SSBB}(\mathcal{B})$ starts in finite time (*Correctness I*).

—Since $\mathcal{SSBB}(\mathcal{B})$ starts, it computes $\mathcal{T}$ as expected (*Correctness II*).

We now present the *BreakingIn* and *$\mathcal{SSBB}$-Friendly* properties that are the properties required in $\mathcal{B}$ so that $\mathcal{SSBB}(\mathcal{B})$ satisfies *Correctness I and II*.

3.1.1 *The BreakingIn Property.* By *Correctness I*, starting from any configuration, the system must reach a configuration in which $\mathcal{SSBB}(\mathcal{B})$ can properly start. This implies that when the root receives a request for an execution of $\mathcal{SSBB}(\mathcal{B})$, $\mathcal{SSBB}(\mathcal{B})$ must start in finite time but without aborting a previously started wave. One way to obtain this property is to design in $\mathcal{B}$ a predicate for r, noted $\mathcal{B}.End(r)$, satisfying the *BreakingIn* property defined in Definition 3.2. It is important to note that such a predicate is not necessarily already used in the protocol to transform. However, it can be defined using the protocol variables and it is essential because it is necessary to obtain snap-stabilization as will be shown in Theorem 3.6.

Beforehand, let us define the *Stable* property, which is a particular characteristic of any predicate satisfying *BreakingIn*.

Definition 3.1 (*Stable*). Let $\mathcal{X}$ be a predicate and $\mathcal{D}$ a daemon. We say that $\mathcal{X}$ satisfies *Stable*($\mathcal{D}$) *iff*: $\mathcal{X}$ is true implies that if there exists some actions guarded by $\mathcal{X}$, then at least one will be executed despite $\mathcal{D}$.

Example: Assume that a predicate $\mathcal{X}$ is satisfied in the configuration γ and there is one action $\mathcal{A}_{\mathcal{X}}$ guarded by $\mathcal{X}$. $\mathcal{X}$ satisfies *Stable*($\mathcal{D}$) means that $\mathcal{A}_{\mathcal{X}}$ is executed in finite time in any suffix of execution starting from γ scheduled by $\mathcal{D}$. Hence, the *stability* induces consequences depending on the fairness of the daemon, for example, if $\mathcal{D} = SF$, then $\mathcal{X}$ is satisfied infinitely often from γ until $\mathcal{A}_{\mathcal{X}}$ is executed.

Definition 3.2 (*BreakingIn*). Let $\mathcal{P}$ be a single-initiator wave protocol where decisions only occur at the initiator. Let $\mathcal{X}(r)$ be a predicate defined on the $\mathcal{P}$ variables of r and its neighbors. Let $\mathcal{D}$ be a daemon. $\mathcal{X}(r)$ satisfies *BreakingIn*($\mathcal{D}$) *iff* $\mathcal{X}(r)$ satisfies these four conditions in any execution of $\mathcal{P}$ scheduled by $\mathcal{D}$:

1. Starting from any configuration, $\mathcal{X}(r)$ is satisfied in finite time.
2. If $\mathcal{X}(r)$ is satisfied and a $\mathcal{P}$ wave was started, then this wave is terminated.
3. From any configuration where $\mathcal{X}(r)$ is satisfied, if there is no external (with regard to $\mathcal{P}$) intervention,[2] all $\mathcal{P}$ actions are disabled at r.
4. $\mathcal{X}(r)$ satisfies *Stable*($\mathcal{D}$).

Example: As shown below, many (self- or snap-) stabilizing fundamental protocols explicitly use a predicate satisfying *BreakingIn*. For example, in any stabilizing *propagation of information with feedback* protocol (PIF protocol), the predicate that characterizes the local state of r (the initiator) when it is waiting for the next message to broadcast satifies *BreakingIn*. Indeed, starting from any configuration, (1) r reaches such a local state in finite time, (2) when r is in this state, any previous PIF is done, (3) in this state, r is waiting for the next message to broadcast, and (4) when the initiator is in this state and there is a new message to broadcast, the daemon cannot prevent the new broadcast from starting.

Conversely, in some stabilizing protocol there is no predicate satisfying *BreakingIn*. For example, Johnen et al. [2002] propose a broadcast (without feedback) protocol where messages are broadcasted in a pipeline manner: r does not wait the termination of the current broadcast to start the next one. In this protocol, there is no predicate satisfying *BreakingIn* because the local state of r is not sufficient to decide if any previous broadcast is terminated.

The following result is a consequence of Claims 1, 3, and 4 of Definition 3.2.

Consequence 3.3. *Let $\mathcal{P}$ be a single-initiator wave protocol where decisions only occur at the initiator. Let $\mathcal{X}(r)$ be a predicate satisfying* BreakingIn($\mathcal{D}$). *Let $\mathcal{D} \in \{SF, WF, UF\}$. Let $e = \gamma_0, \gamma_1, \ldots$ be an execution of $\mathcal{P}$ scheduled by $\mathcal{D}$.*

1. *If $\mathcal{D} = SF$, then, in any non-empty suffix e' of e, $\exists \gamma_i \in e'$ such that $\mathcal{X}(r)$ in γ_i.*
2. *If $\mathcal{D} = WF$, then, $\exists \gamma_i \in e$ such that $\forall j \geq i$, $\mathcal{X}(r)$ in γ_j.*
3. *If $\mathcal{D} = UF$, then, e is finite and $\mathcal{X}(r)$ is satisfied in the terminal configuration.*

We now show that the *BreakingIn* property is necessary to obtain stabilizing single-initiator wave protocols where decisions only occur at the initiator. Hence we will conclude that *BreakingIn* is weaker than stabilization.

Let $\mathcal{P}$ be a (self- or snap-) stabilizing single-initiator wave protocols where decisions only occur at the initiator. Each $\mathcal{P}$ wave (conform or not to the specifications) terminates in finite time by a decision at r. After this decision, the system reaches in finite time a configuration γ from which it is ready to restart: $\mathcal{P}$ is waiting for the next external request. If there is no request, all $\mathcal{P}$ actions

[2]For example, we consider any request for $\mathcal{P}$ as an external intervention.

are disabled at r in γ (remember that the starting actions of $\mathcal{P}$ are triggered by external requests). Let $\mathcal{X}(r)$ be a predicate that characterizes the local state of r in γ. By Definition 3.2, $\mathcal{X}(r)$ satisfies *BreakingIn*($\mathcal{D}$) where $\mathcal{D}$ is the daemon supported by $\mathcal{P}$. Hence:

LEMMA 3.4. *In any stabilizing single-initiator wave protocol where decisions only occur at the initiator, there exists a predicate satisfying* BreakingIn *at the initiator.*

The next lemma shows that the reciprocal of Lemma 3.4 is false.

LEMMA 3.5. *There exist nonstabilizing single-initiator wave protocols where decisions only occur at the initiator in which there exists a predicate at the initiator satisfying* BreakingIn.

PROOF. Two examples of such protocols are given in Section 4. □

By Lemmas 3.4 and 3.5, follows the main motivation of the article:

THEOREM 3.6. *The* BreakingIn *property for single-initiator wave protocol where decisions only occur at the initiator is strictly weaker than stabilization.*

Theorem 3.6 formally states that the *BreakingIn* property is easier to obtain than stabilization. We propose nonstabilizing protocols satisfying *BreakingIn* in Section 4. The code of these protocols are not only close to the basic non fault-tolerant solution but drastically simpler than the existing (self- or snap-) stabilizing versions.

3.1.2 *The SSBB-Friendly Property.* Assume now the existence in $\mathcal{B}$ of a predicate $\mathcal{B}.End(r)$ satisfying *BreakingIn*($\mathcal{D}$). $\mathcal{SSBB}$ can take the execution control of $\mathcal{B}$ in finite time without aborting any previous started $\mathcal{B}$ wave. It remains then to ensure *Correctness II*. To that goal, we use a simple method: Upon a request, we wait until $\mathcal{B}.End(r)$ is satisfied and then reset the $\mathcal{B}$ variables before $\mathcal{B}$ executes a wave. To apply this method, we assume that, for each process p, all the variables assignments required to generate a *normal starting configuration* of $\mathcal{A}$ are stored in a macro of $\mathcal{B}$, noted $\mathcal{B}.Init_p$. We note $\mathcal{B}.Init$ the set of macros $\mathcal{B}.Init_p$ defined for all processes p. Using $\mathcal{B}.Init$, the reset is trivially initiated at the starting action of $\mathcal{SSBB}(\mathcal{B})$ and, as soon as the reset terminates, $\mathcal{B}$ computes the task $\mathcal{T}$ as $\mathcal{A}$ in a non-faulty situation. In particular, this means that the starting actions of $\mathcal{SSBB}(\mathcal{B})$ corresponds to the starting actions of the reset and, of course, $\mathcal{SSBB}(\mathcal{B})$ will take the requests for $\mathcal{B}$ into account instead of $\mathcal{B}$ itself. Hence, we can now define the *SSBB-Friendly* property: The property that $\mathcal{B}$ must satisfy to be snap-stabilized by $\mathcal{SSBB}$.

Definition 3.7 (*SSBB-Friendly*). Let $\mathcal{P}$ be a single-initiator wave protocol where decisions only occur at the initiator. $\mathcal{P}$ satisfies *SSBB-Friendly*($\mathcal{T}$,$\mathcal{D}$) where $\mathcal{T}$ is a task and $\mathcal{D}$ a daemon *iff*:

1. $\mathcal{P}$ contains a predicate $\mathcal{P}.End(r)$ satisfying *BreakingIn*($\mathcal{D}$).
2. $\mathcal{P}$ contains a set of macros $\mathcal{P}.Init$ such that, starting from any configuration generated by $\mathcal{P}.Init$, $\mathcal{P}$ computes $\mathcal{T}$.

3.2 Tool: Snap-Stabilizing PIF

Upon a request, $\mathcal{SSBB}(\mathcal{B})$ resets the $\mathcal{B}$ variables using $\mathcal{B}.Init$ so that the system reaches the *normal starting configuration* of $\mathcal{A}$ and, then, gives the execution control to $\mathcal{B}$ so that it computes $\mathcal{T}$ as $\mathcal{A}$ in a safe environment. A well-known technique to perform a reset is based on the *PIF*. A *PIF* scheme can be specified as follows:

1. If r has a message m to broadcast, r starts the broadcast of m in finite time.
2. Then, every other process acknowledges the receipt of m, such an acknoledgment reaches r in finite time.

Any snap-stabilizing *PIF* protocol satisfies this specification whatever the initial configuration. We distinguish two phases in a *PIF* wave: The *broadcast phase* where the processes receive the message followed by the *feedback phase* where the non-root processes send an acknowledgment for the receipt of m. Using the *PIF* scheme, the reset protocol can be performed as follows:

1. The root process broadcasts an "abort" message m.
2. Upon the reception of m, the processes abort their local execution of $\mathcal{B}$.
3. Finally, the processes reset their $\mathcal{B}$ variables during the feedback phase.

To design $\mathcal{SSBB}$, we need to use a snap-stabilizing *PIF* working even if the daemon is unfair to be able to snap-stabilize protocols working under any kind of daemons. Such a protocol, noted $\mathcal{PIF}$, is provided in Cournier et al. [2006b]. Due to the lack of space we do not present Algorithm $\mathcal{PIF}$ here.

We now explain how to build $\mathcal{SSBB}$ using Algorithm $\mathcal{PIF}$.

3.3 SSBB Protocol

$\mathcal{SSBB}$ is a composite protocol obtained using the composition defined as follows:

Definition 3.8 (*Composition*). The composition $\mathcal{P}_2 \circ_{|\mathcal{G}} \mathcal{P}_1$ of the two protocols $\mathcal{P}_1$ and $\mathcal{P}_2$ is the program satisfying the following four conditions:

1. $\mathcal{P}_2 \circ_{|\mathcal{G}} \mathcal{P}_1$ contains all the variables and actions of $\mathcal{P}_1$ and $\mathcal{P}_2$.
2. $\mathcal{G}$ is a predicate defined on the variables of $\mathcal{P}_1$.
3. Each action of $\mathcal{P}_2$, $\mathcal{L}_i :: \mathcal{H}_i \rightarrow \mathcal{S}_i$, is replaced in $\mathcal{P}_2 \circ_{|\mathcal{G}} \mathcal{P}_1$ by $\mathcal{L}_i :: \mathcal{G} \wedge \mathcal{H}_i \rightarrow \mathcal{S}_i$.
4. A process may be enabled in both $\mathcal{P}_1$ and $\mathcal{P}_2$. In this case, if the daemon chooses it, it executes the priority enabled actions of each $\mathcal{P}_i$ in the same step.

Using the composition, $\mathcal{SSBB}(\mathcal{B}) = \mathcal{B} \circ_{|Ok(p)} Reset(\mathcal{B})$ where $Reset(\mathcal{B})$ is a slightly modified version of $\mathcal{PIF}$ (the code of $Reset(\mathcal{B})$ is given in the appendix, Section A.2) and $Ok(p)$ is a predicate defined on the $\mathcal{PIF}$ for any process p. $Ok(p)$ indicates when p does not participate to any reset (i.e. *PIF* wave): In this case $Ok(p) = true$, otherwise $Ok(p) = false$. $Reset(\mathcal{B})$ is used for resetting the $\mathcal{B}$ variables before any execution of $\mathcal{B}$. To obtain $Reset(\mathcal{B})$ from *PIF*, we just modify the feedback action of $Reset(\mathcal{B})$ so that the $\mathcal{B}$ variables will be reset using $\mathcal{B}.Init$ during the feedback phase. We also add the condition $\mathcal{B}.End(r) = true$

into the guard of any starting action of $Reset(\mathcal{B})$ on r so that we are sure not to reset the $\mathcal{B}$ variables wrongly. We use the predicate $Ok(p)$ into the composition so that each process p stops its local execution of $\mathcal{B}$ from the moment where it receives the abort message until the local termination of the reset at p. Indeed, we already know that p continuously satisfies $\neg Ok(p)$ during its participation to a reset. So, while p participates to a reset, $Ok(p)$ is false and all actions of $\mathcal{B}$ in $\mathcal{SSBB}(\mathcal{B})$ are disabled at p. Finally, as any $\mathcal{SSBB}(\mathcal{B})$ begins by a reset of the $\mathcal{B}$ variables, the starting actions of $\mathcal{SSBB}(\mathcal{B})$ correspond to those of $Reset(\mathcal{B})$ and, as we already said, these actions take the request for $\mathcal{B}$ into account instead of $\mathcal{B}$ itself.

3.4 Correctness

We now show that $\mathcal{SSBB}(\mathcal{B})$ is snap-stabilizing for the specification of the task $\mathcal{T}$ and assuming the daemon $\mathcal{D} \in \{SF, WF, UF\}$.

First, we deduce from Cournier et al. [2006b] the following properties on $Reset(\mathcal{B})$:

PROPERTY 3.9. *From Cournier et al. [2006b], follows:*

1. *After $Reset(\mathcal{B})$ starts a broadcast, the system reaches in finite time a configuration where every process is in the feedback phase associated to this broadcast.*
2. *From any initial configuration, $Reset(\mathcal{B})$ can start in $O(\Delta \times N^3)$ steps and $O(N)$ rounds.*
3. *From any initial configuration, a complete $Reset(\mathcal{B})$ wave costs $O(\Delta \times N^3)$ steps and $O(N)$ rounds.*
4. *From any initial configuration, a process p acknowledges a message not broadcast by* r *at most twice.*
5. *From any configuration where $Ok(p)$, a process p is guaranteed to acknowledge a message broadcast by* r *since its second feedback.*

As $\mathcal{B}$ is designed for solving the specific task $\mathcal{T}$, its code is independent of the variables of $Reset(\mathcal{B})$ and we can make the following assumption:

ASSUMPTION 3.10. *$\mathcal{B}$ does not write into the $Reset(\mathcal{B})$ variables.*

Definition 3.11 (*Fair [Tel 2001]*). An execution e of the composite protocol $\mathcal{P}_2 \circ_{|\mathcal{G}} \mathcal{P}_1$ is *fair* with regard to $\mathcal{P}_i$ ($i \in \{1,2\}$) *iff* if one of these conditions holds: (1) e is finite, (2) e contains infinitely many steps of $\mathcal{P}_i$, or (3) contains an infinite suffix in which no step of $\mathcal{P}_i$ is enabled.

THEOREM 3.12. *$\mathcal{SSBB}(\mathcal{B})$ is a fair composition of $Reset(\mathcal{B})$ and $\mathcal{B}$.*

PROOF. Assume, by the contradiction, that there exists at least one execution e of $\mathcal{SSBB}(\mathcal{B})$ which is not *fair*. Then, e contains infinitely many steps and two cases are possible according to Definition 3.11:

1. There exists an infinite suffix e' of e where some actions of $\mathcal{B}$ are enabled but only actions of $Reset(\mathcal{B})$ are executed. Then, e' contains infinitely many

steps of $Reset(\mathcal{B})$. Now, by Claim 3 of Property 3.9, each $Reset(\mathcal{B})$ wave is executed in a finite number of actions. So, $Reset(\mathcal{B})$ executes waves infinitely often in e'. $Reset(\mathcal{B})$ behaves as $\mathcal{PIF}$ which is snap-stabilizing. So, during the first wave in e', $Reset(\mathcal{B})$ executes a complete feedback phase. Now, when r feedbacks, r resets its $\mathcal{B}$ variables using $\mathcal{B}.Init_r$. After this local reset, $\mathcal{B}.End(r)$ becomes false and, as the $\mathcal{B}$ variables are no more modified (by assumption, no action of $\mathcal{B}$ are executed in e'), $\mathcal{B}.End(r)$ is false forever. This implies that the starting action of $Reset(\mathcal{B})$ becomes disabled forever. Hence, $Reset(\mathcal{B})$ cannot execute waves infinitely often in e', a contradiction.

2. There exists an infinite suffix e' of e where some actions of $Reset(\mathcal{B})$ are enabled but only actions of $\mathcal{B}$ are executed. Then, e' contains infinitely many steps of $\mathcal{B}$. As $\mathcal{B}$ does not write into the $Reset(\mathcal{B})$ variables (Assumption 3.10) and no action of $Reset(\mathcal{B})$ are executed in e'. All actions of $Reset(\mathcal{B})$ that are enabled in the first configuration of e' are continuously enabled and, by the contradiction, $\mathcal{D} = UF$: Only the unfair daemon can prevent forever a continuously enabled action to be executed. Now, Claim 3 of Consequence 3.3 implies that the system reaches in finite time a configuration γ_j where only actions of $Reset(\mathcal{B})$ are enabled. Hence, at least one enabled action of $Reset(\mathcal{B})$ is executed in $\gamma_j \mapsto \gamma_{j+1}$, a contradiction. □

By Theorem 3.12, Assumption 3.10, and Claim 2 of Property 3.9, follows:

THEOREM 3.13. *From any initial configuration, $\mathcal{SSBB}(\mathcal{B})$ can start in finite time.*

THEOREM 3.14. *From any configuration where* r *starts $\mathcal{SSBB}(\mathcal{B})$, the task $\mathcal{T}$ is computed according to its specifications.*

PROOF. $\mathcal{SSBB}(\mathcal{B})$ starts by a $Reset(\mathcal{B})$ wave. r then satisfies $\neg Ok(r)$ until the $Reset(\mathcal{B})$ wave terminates. So, during the $Reset(\mathcal{B})$ wave, r cannot executes any action of $\mathcal{B}$. In particular, r cannot initiate $\mathcal{B}$ before $Reset(\mathcal{B})$ terminates.

By Theorem 3.12 and Assumption 3.10, $\mathcal{B}$ does not perturb the behavior of $Reset(\mathcal{B})$. Now, by Claim 1 of Property 3.9, after r starts a broadcast, the system reaches in finite time a configuration γ_i where each process is in the feedback phase associated to the broadcast of r. As the $\mathcal{B}$ variables are locally reset at p when a process p feedbacks and no action of $\mathcal{B}$ is executed at p while p participates to the $Reset(\mathcal{B})$ wave ($\neg Ok(p)$), γ_i corresponds to the configuration generated by $\mathcal{B}.Init$ (i.e., a *normal initial configuration* of $\mathcal{A}$). Moreover, owing the fact that the system reaches γ_i results in two other consequences: (1) From γ_i, no process executes any action of $\mathcal{B}$ before r initiates $\mathcal{B}$, and (2) from γ_i, the system contains no abnormal behavior related to $Reset(\mathcal{B})$. Hence, when $Reset(\mathcal{B})$ terminates at r, the system is in a configuration γ_j where $\forall p \in V$, $Ok(p)$ (a *normal initial configuration* of $\mathcal{PIF}$), the $\mathcal{B}$ variables describe the configuration generated by $\mathcal{B}.Init$, and $\mathcal{B}.End(r) = false$. From γ_j, no reset will be initiated before that $\mathcal{B}.End(r)$ is satisfied, that is, before the decision associated to the execution of $\mathcal{B}$ that r will start. So, by Claim 2 of Definition 3.7, from γ_j, $\mathcal{B}$ executes the task $\mathcal{T}$ according to its specifications. □

By Remark 2.3, Theorems 3.13 and 3.14, follows:

THEOREM 3.15. *$\mathcal{SSBB}(\mathcal{B})$ is snap-stabilizing for the specification of $\mathcal{T}$ under the daemon $\mathcal{D}$.*

3.5 Complexity Analysis

We first evaluate the round complexity of $\mathcal{SSBB}(\mathcal{B})$. To that goal, we introduce the following notation: let $R(\mathcal{B})$ be the number of rounds that $\mathcal{B}$ requires so that $\mathcal{B}.End(r)$ is continuously satisfied starting from any configuration. The first lemma gives the delay of $\mathcal{SSBB}(\mathcal{B})$ in rounds.

THEOREM 3.16. *From any initial configuration, $\mathcal{SSBB}(\mathcal{B})$ starts in $O(N + R(\mathcal{B}))$ rounds.*

PROOF. Let $Broadcast(r)$ be the guard of the starting action of *PIF*. By Claim 2 of Property 3.9, starting from any configuration, $\mathcal{SSBB}(\mathcal{B})$ reaches in $O(N)$ rounds a configuration from which $Broadcast(r)$ is continuously satisfied until $\mathcal{SSBB}(\mathcal{B})$ starts. After at most $R(\mathcal{B})$ additional rounds, $\mathcal{B}.End(r)$ continuously holds and, as a consequence, $\mathcal{SSBB}(\mathcal{B})$ starts during the next round. □

The worst case of Theorem 3.16 corresponds to the number of rounds necessary to perform a complete $\mathcal{SSBB}(\mathcal{B})$ wave except its first step. Hence, the round complexity of a complete $\mathcal{SSBB}(\mathcal{B})$ wave is of the same order as the round complexity of the delay to start a $\mathcal{SSBB}(\mathcal{B})$ wave. Hence, follows:

COROLLARY 3.17. *From any initial configuration, a $\mathcal{SSBB}(\mathcal{B})$ wave is executed in $O(N + R(\mathcal{B}))$ rounds.*

We now evaluate the step complexity of $\mathcal{SSBB}(\mathcal{B})$. To that goal we introduce the following notation: Let $S(\mathcal{B})$ be the maximal number of actions that $\mathcal{B}$ can execute before to reach a terminal configuration with respect to the $\mathcal{B}$ variables.

THEOREM 3.18. *From any initial configuration, $\mathcal{SSBB}(\mathcal{B})$ starts in $O(\Delta \times N^3 + S(\mathcal{B}) \times N)$ steps.*

PROOF. By Claim 2 of Property 3.9, $O(\Delta \times N^3)$ actions of $Reset(\mathcal{B})$ are executed before $\mathcal{SSBB}(\mathcal{B})$ starts. We now evaluate the number of $\mathcal{B}$ actions that are executed before $\mathcal{SSBB}(\mathcal{B})$ starts. First, $Reset(\mathcal{B})$ writes into the $\mathcal{B}$ variables only during the feedback phase and any process (N) executes at most two corrupted feedbacks before $\mathcal{SSBB}(\mathcal{B})$ starts (by Claim 4 of Property 3.9). Then, at most $S(\mathcal{B})$ actions of $\mathcal{B}$ are executed between each action of $Reset(\mathcal{B})$ that writes into the $\mathcal{B}$ variables: Between each feedback. So, $O(S(\mathcal{B}) \times N)$ actions of $\mathcal{B}$ are executed before $\mathcal{SSBB}(\mathcal{B})$ starts. Hence, the delay of $\mathcal{SSBB}(\mathcal{B})$ is in $O(\Delta \times N^3 + S(\mathcal{B}) \times N)$ steps. □

Similar to Theorem 3.16, Theorem 3.18 implies the following corollary:

COROLLARY 3.19. *From any initial configuration, a $\mathcal{SSBB}(\mathcal{B})$ wave is executed in $O(\Delta \times N^3 + S(\mathcal{B}) \times N)$ steps.*

4. APPLICATIONS

4.1 Depth-First Token Circulation

In arbitrary rooted networks, a Depth-First Token Circulation (noted *DFTC*) works as follows: A token is first created at the root and, then, passed from one neighbor to another in a depth-first order such that every process eventually gets it in one single circulation. We now propose a protocol called $\mathcal{DFS}$ (cf. Algorithms 1 and 2) satisfying $\mathcal{SSBB}$-*Friendly*(*DFTC*,*UF*): Combining $\mathcal{DFS}$ with $\mathcal{SSBB}$, we trivially obtain a snap-stabilizing *DFTC* protocol, $\mathcal{SSBB}(\mathcal{DFS})$.

Let γ_α be the configuration generated by $\mathcal{DFS}$.*Init*: $\forall p \in V$, $Succ_p = idle$ in γ_α. We first show that $\mathcal{DFS}$ executes a *DFTC* starting from γ_α. In γ_α, *Fwd-action* at r is the only enabled action. So, r creates a token by *Fwd-action* in $\gamma_\alpha \mapsto \gamma_{\alpha+1}$: Using $Next_r$, r points out with $Succ_r$ its minimal neighbor by $\prec_r$ meaning that it is its *successor*. Then, at each configuration and until the circulation terminates, exactly one process p is enabled and two cases are possible for p:

1. p satisfies $(Succ_p = idle) \wedge (\exists q \in Neig_p :: Succ_q = p)$. In this case, q designated $p \neq r$ and p executes $Fwd\text{-}action$ to receive the token. By $Fwd\text{-}action$, p chooses a successor, if any. For this task, two cases are possible:
   a) $\forall p' \in Neig_p$, $Succ_{p'} \neq idle$, i.e., all its neighbors have been visited. So, p sets $Succ_p$ to *done*, *i.e.*, the circulation from p is terminated.
   b) Otherwise, p designates its minimal neighbor satisfying $Succ = idle$.
2. p satisfies $Succ_p = q$ such that $(q \in Neig_p \wedge Succ_q = done)$, *i.e.*, q was the successor of p but the circulation from q is terminated. Then, p backtracks the token to continue the circulation using another neighbor which is still not visited, if any. So, p executes *Bwd-action* to choose a new successor as previously.

By this mechanism, the circulation *visits* all processes in a depth-first order and r executes $Succ_r = done$ meaning that the circulation is terminated after $2N - 1$ steps. Hence, follows:

LEMMA 4.1. *From γ_α, $\mathcal{DFS}$ executes a DFTC in $2N - 1$ steps (resp. rounds).*

By Lemma 4.1, we know that $\mathcal{DFS}$ satisfies Claim 2 of the definition of $\mathcal{SSBB}$-*Friendly* (Definition 3.7) for a distributed unfair daemon. Now, to show that $\mathcal{DFS}.End(r)$ satisfies *BreakingIn*(*UF*) (Claim 1 of Definition 3.7), we have to prove that, starting from any configuration, $\mathcal{DFS}$ converges in a finite number of steps to a terminal configuration where $\mathcal{DFS}.End(r)$ is satisfied (cf. Consequence 3.3). The behavior of $\mathcal{DFS}$ from a configuration different from γ_α is the following: When a process p locally detects that the system is not in a configuration reachable from γ_α, it definitively sets $Succ_p$ to *done* (*Err-action*). Below, we will show that even if the system starts from a configuration different from γ_α, the system reaches a terminal configuration where $\mathcal{DFS}.End(r)$ is true in a finite number of steps. We prove this fact in two steps: (1) We first show that every execution of $\mathcal{DFS}$ contains a finite number of steps; (2) we then show that $\mathcal{DFS}.End(r)$ is satisfied in any terminal configuration.

Algorithm 1. $\mathcal{DFS}$ for $p = r$

Input: $Neig_p$: set of neighbors (locally ordered);
Variable: $Succ_p \in Neig_p \cup \{idle, done\}$;
Macros:
$\boldsymbol{Init}_p = Succ_p := idle;$
$Pred_p = \{q \in Neig_p :: Succ_q = p\};$
$Next_p = (q = \min_{\prec_p}\{q' \in Neig_p :: Succ_{q'} = idle\})$ **if** q **exists**, $done$ **otherwise**;
Predicates:
$\boldsymbol{End(p)} \equiv (Succ_p = done)$
$Forward(p) \equiv (Succ_p = idle)$
$Backward(p) \equiv (\exists q \in Neig_p :: Succ_p = q \wedge Succ_q = done)$
$Error(p) \equiv (Succ_p \neq done) \wedge [(|Pred_p| \neq 0) \vee ((Succ_p = idle) \wedge (\exists q \in Neig_p :: Succ_q \neq idle))]$
Actions:

$Err\text{-}action$	::	$Error(p)$	$\rightarrow$	$Succ_p := done;$
$Fwd\text{-}action$	::	$Forward(p)$	$\rightarrow$	$Succ_p := Next_p;$
$Bwd\text{-}action$	::	$Backward(p)$	$\rightarrow$	$Succ_p := Next_p;$

LEMMA 4.2. *Starting from any configuration, $\mathcal{DFS}$ reaches a terminal configuration in at most $(\Delta + 1) \times N$ steps.*

PROOF. It is easy to see that each process (N) designates each of its neighbors at most once (Δ actions) before definitively sets $Succ_p$ to *done* (one action). □

LEMMA 4.3. *In any terminal configuration of $\mathcal{DFS}$,* r *satisfies $\mathcal{DFS}.End(r)$.*

PROOF. We prove this lemma by checking every configuration of $\mathcal{DFS}$ where $\mathcal{DFS}.End(r)$ is not satisfied. We can then show that there always exist some enabled actions in such configurations. □

By Lemmas 4.2 and 4.3, $\mathcal{DFS}.End(r)$ satisfies Claims 1 and 4 of the *BreakingIn* property under a distributed unfair daemon. Also, from γ_α, $\mathcal{DFS}.End(r)$ is satisfied only when the token circulation is done (Claim 2 of *BreakingIn*). Finally, by checking the actions of Algorithms 1 and 2, we can deduce that $\mathcal{DFS}.End(r)$ satisfies Claim 3 of *BreakingIn*. Hence, follows:

LEMMA 4.4. *$\mathcal{DFS}.End(r)$ satisfies BreakingIn(UF).*

Algorithm 2. $\mathcal{DFS}$ for $p \neq r$

Input: $Neig_p$: set of neighbors (locally ordered);
Variable: $Succ_p \in Neig_p \cup \{idle, done\}$;
Macros:
$\boldsymbol{Init}_p = Succ_p := idle;$
$Pred_p = \{q \in Neig_p :: Succ_q = p\};$
$Next_p = (q = min_{\prec_p}\{q' \in Neig_p :: Succ_{q'} = idle\})$ **if** q **exists**, $done$ **otherwise**;
Predicates:
$Forward(p) \equiv (|Pred_p| = 1) \wedge (Succ_p = idle)$
$Backward(p) \equiv (\exists q \in Neig_p :: Succ_p = q \wedge Succ_q = done)$
$Error(p) \equiv (Succ_p \neq done) \wedge [(|Pred_p| > 1) \vee (Succ_p \neq idle \wedge |Pred_p| = 0) \vee ((Succ_p = idle) \wedge (\exists q \in Neig_p :: Succ_q = done))]$
Actions:

$Err\text{-}action$	::	$Error(p)$	$\rightarrow$	$Succ_p := done;$
$Fwd\text{-}action$	::	$Forward(p)$	$\rightarrow$	$Succ_p := Next_p;$
$Bwd\text{-}action$	::	$Backward(p)$	$\rightarrow$	$Succ_p := Next_p;$

By Lemmas 4.1 and 4.4, we have:

LEMMA 4.5. *$\mathcal{DFS}$ satisfies $\mathcal{SSBB}$-Friendly(**DFTC,UF**).*

By Lemma 4.5 and Theorem 3.15, we have:

THEOREM 4.6. *$\mathcal{SSBB}(\mathcal{DFS})$ is a snap-stabilizing DFTC protocol assuming a distributed unfair daemon.*

We already proposed in Cournier et al. [2004] and Cournier et al. [2005] two snap-stabilizing *DFTC* protocols working under a distributed unfair daemon. The code of $\mathcal{DFS}$ is simpler than the code of these solutions as well as any self-stabilizing *DFTC*. Actually, the code $\mathcal{DFS}$ is similar to the one of the basic non-fault-tolerant solution. The complexity analysis provided below will reveal that $\mathcal{SSBB}(\mathcal{DFS})$ is also more efficient than the previous solutions.

4.1.1 *Complexity Analysis of $\mathcal{SSBB}(\mathcal{DFS})$*

LEMMA 4.7. *Starting from any configuration, $\mathcal{DFS}$ reaches a terminal configuration in at most $2N-1$ rounds.*

PROOF. By Lemma 4.1, from γ_α, the system reaches a terminal configuration in $2N-1$ rounds. If the initial configuration is different from γ_α, the corrections are performed in parallel using *Err-action* and, at each correction, the corrected process becomes disabled forever. Thus, the corrections allow to save visits and the worst case to reach a terminal configuration actually corresponds to an execution starting from γ_α. □

We now compare the complexity of our solution with the previous solutions. The main drawback of the solution in Cournier et al. [2004] is its memory requirement: $O(N \times \log N)$ bits per process. In Cournier et al. [2005], we solve this drawback by proposing a protocol using $O(\log N)$ bits per process. Here, the memory requirement of $\mathcal{DFS}$ is in $O(\log \Delta)$ bits per process and the one of $Reset(\mathcal{B})$ is in $O(\log N)$ bits per process. Hence, $\mathcal{SSBB}(\mathcal{DFS})$ is as efficient in memory as the best already proposed solution [Cournier et al. 2005], that is, $O(\log N)$ bits per process. The snap-stabilizing *DFTC* protocol provided in Cournier et al. [2004] is very efficient in time complexity: It performs a single token circulation in $O(N)$ rounds and $O(N^2)$ steps (the delay is of the same order). The protocol provided in Cournier et al. [2005] is less efficient: It performs a single token circulation in $O(N^2)$ rounds and $O(\Delta \times N^3)$ steps (the delay is of the same order). The delay of $\mathcal{SSBB}(\mathcal{DFS})$ is in $O(N)$ rounds (by Theorem 3.16 Lemma 4.7) and $O(\Delta \times N^3)$ steps (by Theorem 3.18 and Lemma 4.2). Then, from any configuration, a depth-first token circulation is executed in $O(N)$ rounds (by Corollary 3.16, Lemmas 4.1 and 4.7) and $O(\Delta \times N^3)$ steps (by Corollary 3.18 and Lemma 4.2). So, the round complexity of $\mathcal{SSBB}(\mathcal{DFS})$ matches the one of Cournier et al. [2004] and the steps complexity of $\mathcal{SSBB}(\mathcal{DFS})$ matches the one of Cournier et al. [2005]. Thus, $\mathcal{SSBB}(\mathcal{DFS})$ is a good trade-off between the two previous solutions.

4.2 Breath-first Spanning Tree Construction

Algorithm 3. $\mathcal{BFS}$ for $p = r$

Input: $Neig_p$: set of neighbors (locally ordered);
Constants: $Par_p = \perp$; $Level_p = 0$;
Variable: $Status_p \in \{C,R,F,D\}$;
Macros:
$\boldsymbol{Init}_p \quad = Status_p := C$;
$Children_p = \{q \in Neig_p :: Status_q \neq C \wedge Par_q = p\}$;
Predicates:
$\boldsymbol{End(p)} \equiv (Status_p = D)$
$Finish(p) \equiv (\forall q \in Children_p :: Status_q = D)$
$NPhase(p) \equiv (Status_p = F) \wedge [\forall q \in Neig_p :: (Status_q \neq C) \wedge (q \in Children_p \Rightarrow Status_q \in \{B,D\})]$
$Start(p) \equiv (Status_p = C) \wedge (\forall q \in Neig_p :: Status_q \in \{C,D\})$
$ROK(p) \equiv (Status_p = R) \wedge (\forall q \in Children_p :: Status_q \in \{R,D\})$
$Forward(p) \equiv Start(p) \vee ROk(p)$
Actions:

New-action	::	$NPhase(p) \wedge \neg Finish(p)$	$\rightarrow$	$Status_p := R$;
Fwd-action	::	$Forward(p)$	$\rightarrow$	$Status_p := F$;
D-action	::	$NPhase(p) \wedge Finish(p)$	$\rightarrow$	$Status_p := D$;

In arbitrary rooted networks, a Breath-First Spanning Tree Construction, noted *BFSTC*, consists in the computation of a rooted tree $Tree(r)$ containing all processes and such that the distance of any process along the tree to the root is the smallest one. There exists many self-stabilizing *BFSTC* protocols [Huang and Chen 1992; Johnen 1997]. Delaët et al. [2005]; propose a general scheme allowing in particular to implement a self-stabilizing *BFSTC*. The protocols proposed in Johnen [1997] and Delaët et al. [2005] work under a distributed unfair daemon. However, except Johnen [1997], all these protocols are silent, that is, using such protocols, the system converges to a fix point. The drawback of such protocols is that no process is able to detect the termination of the execution. The protocol proposed in Johnen [1997] is a self-stabilizing *BFSTC* wave protocol. In this protocol, each wave terminates at the root. However, as this protocol is not snap-stabilizing, the root cannot detect when the system verifies its specifications. As a consequence, when a wave terminates, the root cannot detect if a breath-first spanning tree is available. Finally, the transformer in Cournier et al. [2003] allows to design a snap-stabilizing *BFSTC* wave protocol by combining any non fault-tolerant *BFSTC* wave protocol [Awerbuch and Gallager 1985] with their transformer. An important advantage of this latter solution is that the root is now able to detect when a breath-first spanning tree is available: Since the termination of the first wave. However, the solutions obtained with this transformer works at most with a weakly fair daemon. We now design a snap-stabilizing *BFSTC* wave protocol working with a distributed unfair daemon. To that goal, we propose a protocol satisfying $\mathcal{SSBB}$-*Friendly*(*BFSTC*,*UF*) called $\mathcal{BFS}$. The code of the protocol (cf. Algorithms 3 and 4) is close to the non-fault-tolerant solution proposed in Awerbuch and gallager [1985].

Let γ_α be the configuration generated by $\mathcal{BFS}.Init$: $\forall p \in V$, $Status_p = C$ in γ_α. We first show that $\mathcal{BFS}$ builds a breath-first spanning tree from γ_α.

From γ_α, $\mathcal{BFS}$ builds a breath-first spanning tree rooted at r by phases: During the k^{th} phase, all processes at distance k from r hook on to the tree. After the hooking of these processes, r detects in finite time the termination of the phase and then starts another phase if the construction is not terminated. At the beginning of the first phase, *Fwd-action* at r is the only enabled action: r executes $Status_r := F$ in $\gamma_\alpha \mapsto \gamma_{\alpha+1}$. Then, the *Hk-action* of each neighbor of r, p, becomes enabled. By executing $Hk\text{-}action$, each process p hooks on to the tree rooted at r: $Status_p$ is set to B, Par_p points out to r, and $Level_p$ is set to 1. When all neighbors of r have joined the tree, r satisfies $NewPhase(r) \wedge \neg Finish(r)$, that is, the first phase is done and r becomes enabled to start a new phase by *New-action*. r begins the second phase by resetting the *Status* variables in its tree: $Status_r$ is set to R. Then, the neighbors of r also executes $New\text{-}action$ to propagate the reset. When all the neighbors of r have set their $Status$ variable to R, r broadcasts the F value along the tree ($Fwd\text{-}action$) meaning that the neighbors at distance 2 have now to hook on to the tree. When a neighbor of r, p, receives a F value, two cases are possible:

1. p has no neighbor at distance 2 from the root. Then, p satisfies $Backtrack(p) \wedge Finish(p)$ and sets $Status_p$ to D using *D-action* meaning that its subtree is completely built.
2. p has at least one neighbor, q, at distance 2 from the root. Then, q joins the tree by hooking on to the minimal process satisfying $Status = F$ (*Hk-action*). When p detects that all its neighbors have joined the tree, p sets $Status_p$ to B using $Bck\text{-}action$.

When all the neighbors of r have switched their *Status* variable to B or D, r detects the end of the second phase and becomes enabled to started a new phase. Inductively, the other phases work as follows. r is enabled to start the k^{th} phase ($k > 2$) when $Status_r = F \wedge \forall p \in Neig_r, Status_p \in \{B,D\}$. The k^{th} phase begins with a reset initiated by r (*New-action*). When $\forall p \in Neig_r$, $Status_p \in \{R,D\}$, r broadcasts a new F value into the tree meaning that the processes at distance k from the root have now to join the tree. The F value is propagated into the tree as follows: When a process has its parent, Par_q, such that $Status_{Par_q} = F$, q waits until all its children (if any) satisfies $Status \in \{R,D\}$ and, then, broadcasts the F value to its children (*cf. Forward*(q)). During this broadcast, one of these two cases appears:

1. A F value reaches a process p such that all its children satisfy $Status = D$ (i.e., $Backtrack(p) \wedge Finish(p)$). p then sets $Status_p$ to D (*D-action*) meaning that its subtree is completely built.
2. A F value reaches a leaf f of the tree. If f has no neighbor at distance k from the root, f satisfies $Backtrack(p) \wedge Finish(p)$ and sets $Status_f$ to D (*D-action*). Otherwise, f has at least one neighbor, q, such that q is at distance k from r. In this case, each q joins the tree by hooking on to a process of the tree such that $Status = F$ (*Hk-action*). When f detects that all its neighbors are in the tree ($Backtrack(f) \wedge \neg Finish(f)$), f sets $Status_f$ to B ($Bck\text{-}action$).

Algorithm 4. $\mathcal{BFS}$ for $p \neq r$

Input: $Neig_p$: set of neighbors (locally ordered);
Variables: $Status_p \in \{C,R,F,B,D\}$; $Par_p \in Neig_p$; $Level_p \in \mathrm{N}$;
Macros:

$\boldsymbol{Init}_p$	$= Status_p := C;$
$Children_p$	$= \{q \in Neig_p :: Status_q \neq C \wedge Par_q = p\};$
$Potential_p$	$= \{q \in Neig_p :: Status_q = F\};$
$MinPotential_p$	$= \min(\{Level_q :: q \in Potential_p\});$

Predicates:

$Finish(p)$	$\equiv (\forall q \in Children_p :: Status_q{=}D)$
$Leaf(p)$	$\equiv [\forall q \in Neig_p :: (Par_q{=}p) \Rightarrow (Status_q \in \{C,D\})]$
$Hook(p)$	$\equiv (Status_p{=}C){\wedge}(\exists q \in Neig_p :: Status_q{=}F) \wedge Leaf(p)$
$NPhase(p)$	$\equiv (Status_p{=}B){\wedge}(Status_{Par_p}{=}R) \wedge (\forall q \in Children_p :: Status_q \in \{B,D\})$
$Forward(p)$	$\equiv (Status_p{=}R){\wedge}(Status_{Par_p}{=}F) \wedge (\forall q \in Children_p :: Status_q \in \{R,D\})$
$Back(p)$	$\equiv (Status_p{=}F){\wedge}[\forall q{\in}Neig_p{::}(Status_q{\neq}C){\wedge}(q{\in}Children_p$ $\Rightarrow Status_q{\in}\{B,D\})]$
$GoodR(p)$	$\equiv (Status_p{=}R) \Rightarrow (Status_{Par_p} \in \{R,F\})$
$GoodF(p)$	$\equiv (Status_p{=}F) \Rightarrow (Status_{Par_p}{=}F)$
$GoodB(p)$	$\equiv (Status_p{=}B) \Rightarrow (Status_{Par_p} \notin \{C,D\})$
$GoodLevel(p)$	$\equiv (Status_p \neq C) \Rightarrow (Level_p{=}Level_{Par_p} + 1)$
$Bad(p)$	$\equiv \neg Good\,R(p) \vee \neg Good\,F(p) \vee \neg Good\,B(p) \vee \neg Good\,Level(p)$
$Error(p)$	$\equiv (Status_p \neq D) \wedge Bad(p)$

Actions:

Err-action	::	$Error(p)$	$\rightarrow Status_p := D;$
Hk-action	::	$Hook(p)$	$\rightarrow Status_p := B; Par_p := \min_{\prec_p}(Potential_p);$ $Level_p := Level_{Par_p} + 1;$
New-action	::	$NPhase(p)$	$\rightarrow Status_p := R;$
Fwd-action	::	$Forward(p)$	$\rightarrow Status_p := F;$
Bck-action	::	$Back(p) \wedge \neg Finish(p)$	$\rightarrow Status_p := B;$
D-action	::	$Back(p) \wedge Finish(p)$	$\rightarrow Status_p := D;$

The B and D values are then propagated into the tree (using *Bck-* and *D-action*) and r eventually satisfies $Status_r = F \wedge \forall p \in Neig_r, Status_p \in \{B,D\}$ again. If $\exists p \in Neig_r$ such that $Status_p = B$, r starts a $(k+1)^{th}$ phase (*New-action*). Otherwise, the spanning tree is completely built and r $Status_r$ to D (*D-action*) meaning that the *BFSTC* wave is terminated.

By definition, the height of a breath-first spanning tree is equal to D where D is the diameter of the network. So, from γ_α, $\mathcal{BFS}$ execute exactly D phases to build the tree. Also, each phase is performed in $O(D)$ rounds and $O(N)$ steps. Hence:

LEMMA 4.8. *Starting from γ_α, $\mathcal{BFS}$ builds a breath-first spanning tree in $O(D \times N)$ steps and $O(D^2)$ rounds.*

By Lemma 4.8, $\mathcal{BFS}$ satisfies Claim 2 of *$\mathcal{SSBB}$-Friendly* under a distributed unfair daemon. We now show that $\mathcal{BFS}.End(r)$ satisfies $BreakingIn(UF)$ (Claim 1 of *$\mathcal{SSBB}$-Friendly*). To that goal, we have to prove that starting from any configuration $\mathcal{BFS}$ converges in a finite number of steps into a terminal configuration where $\mathcal{BFS}.End(r)$ is satisfied (cf. Consequence 3.3). The behavior of $\mathcal{BFS}$ starting from a configuration different from γ_α is similar to the one of $\mathcal{DFS}$: When a process p locally detects that the system is not in a configuration

reachable from γ_α, it definitively sets $Status_p$ to D ($Err\text{-}action$). So, using the same method as for Lemmas 4.2 and 4.3, we can show the two following lemmas:

LEMMA 4.9. *Starting from any configuration, $\mathcal{BFS}$ reaches a terminal configuration in $O(D \times N)$ steps.*

LEMMA 4.10. *In any terminal configuration of $\mathcal{BFS}$,* r *satisfies $\mathcal{BFS}.End(r)$.*

By Lemmas 4.9 and 4.10, $\mathcal{BFS}.End(r)$ satisfies Claims 1 and 4 of *BreakingIn* under a distributed unfair daemon. Moreover, it is clear that from γ_α, $\mathcal{BFS}.End(r)$ is satisfied only if a breath-first spanning tree is available (Claim 2 of *BreakingIn*). Finally, by checking the actions of Algorithm 3, we can state that $\mathcal{BFS}.End(r)$ satisfies Claim 3 of *BreakingIn*. Hence, follows:

LEMMA 4.11. *$\mathcal{BFS}.End(r)$ satisfies BreakingIn(UF).*

By Lemmas 4.8 and 4.11, we have:

LEMMA 4.12. *$\mathcal{BFS}$ satisfies $\mathcal{SSBB}$-Friendly(BFSTC,UF).*

By Lemma 4.12 and Theorem 3.15, we have:

THEOREM 4.13. *$\mathcal{SSBB}(\mathcal{BFS})$ is a snap-stabilizing BFSTC wave protocol under a distributed unfair daemon.*

The code of $\mathcal{BFS}$ is close to the code of the solution of Awerbuch and Gallager [1985]. Also, it is simpler than the code of Johnen [1997]. Below, the complexity analysis show that $\mathcal{SSBB}(\mathcal{BFS})$ is also efficient.

4.2.1 *Complexity Analysis of $\mathcal{SSBB}(\mathcal{BFS})$.* $\mathcal{BFS}$ use $O(\log N)$ bits per process and $Reset(\mathcal{B})$ $O(\log N)$ bits per process. So, $\mathcal{SSBB}(\mathcal{BFS})$ requires $O(\log N)$ bits per process. Consider now the time complexity. We begin this analysis with the following lemma. It allows to evaluate the round complexity of $\mathcal{SSBB}(\mathcal{BFS})$.

LEMMA 4.14. *Starting from any configuration, $\mathcal{BFS}$ reaches a terminal configuration in $O(D^2)$ rounds.*

PROOF. The proof is similar to the one of Lemma 4.7 (page 15). □

The delay to start a breath-first spanning tree construction is in $O(D^2 + N)$ rounds (by Theorem 3.16 and Lemma 4.14) and $O(\Delta \times N^3)$ steps (by Theorem 3.18 and Lemma 4.9). Finally, starting from any configuration, a breath-first spanning tree is computed in $O(D^2 + N)$ rounds (by Corollary 3.16, Lemmas 4.8 and 4.14) and $O(\Delta \times N^3)$ steps (by Corollary 3.18 and Lemma 4.9).

5. EXTENTION: MUTUAL EXCLUSION

$\mathcal{SSBB}(\mathcal{DFS})$ allows to perform a single snap-stabilizing depth-first token circulation. The snap-stabilizing property guarantees that, starting from any configuration and upon a request, a token is created in finite time and this token visits all the processes in a depth-first order. $\mathcal{SSBB}(\mathcal{DFS})$ is also able to perform a perpetual token circulation, that is, an infinite repetition of token circulation cycles. To that end, we just have to assume that the system is continuously

requesting the token circulation. A common application of perpetual token circulations is the *mutual exclusion*. In the *mutual exclusion*, the existence of a special section of code, called *critical section* (noted $\langle CS \rangle$), is assumed. The critical section must be executed by at most one process at a time. The mutual exclusion can be specified as follows Villain [2002]: Any process that requests $\langle CS \rangle$ enters in $\langle CS \rangle$ in finite time (*liveness*), and if a requesting process enters in $\langle CS \rangle$, it executes $\langle CS \rangle$ alone (*safety*). Note that, in a safe environment, the safety property is equivalent to the property "Never more that one process executes $\langle CS \rangle$ simultaneously" in a sense that any protocol satisfying the first property also satisfies the second one and reciprocally. Indeed, in a safe environment, $\langle CS \rangle$ is initially free and only the requestors can enter into $\langle CS \rangle$. Of course, it is not the case in a faulty environment.

Using any perpetual token circulation, the mutual exclusion can be solved as follows: A process executes its critical section only when it holds a token. Unfortunately, $\mathcal{SSBB}(\mathcal{DFS})$ is not snap-stabilizing for the mutual exclusion. Indeed, starting from any configuration, several corrupted tokens may circulate simultaneously. As a consequence, processes may request and then execute the critical section simultaneously. However, starting from any configuration, any cycle of $\mathcal{SSBB}(\mathcal{DFS})$ is composed of a snap-stabilizing reset of the $\mathcal{DFS}$ variables followed by a single token circulation. So, after the first reset, the specification of the mutual exclusion is verified forever and the protocol is self-stabilizing for the mutual exclusion. Also, r can detect when a token is unique: Once r starts the protocol, the next token that r creates is unique. The next theorem shows that there is a way for any process (not only for r) to detect when they hold a token which is unique.

THEOREM 5.1. *Starting from any configuration, a process p is guaranteed to receive a token that is unique since it executes Fwd-action for the third time.*

PROOF. To execute $Fwd\text{-}action$, a process p must satisfy $Succ_p = idle$ in $\mathcal{DFS}$ and $Ok(p)$ in $Reset(\mathcal{DFS})$. After executing *Fwd-action*, p satisfies $Succ_p \neq idle$. We can remark that p will satisfy $Succ_p = idle$ again only after executing the feedback of $Reset(\mathcal{DFS})$. So, p executes at least one feedback between each execution of *Fwd-action* and, from the third *Fwd-action*, at least two feedbacks have been already executed by p. Now, since the execution of two feedbacks after the first *Fwd-action*, the feedbacks executed by p always correspond to feedbacks of the abort messages broadcasted by r (Claim 5 of Property 3.9, page 10). After such feedbacks, each token that p will receive is unique (the reset is snap-stabilizing). □

By Theorem 5.1, we can trivially obtain an efficient self-stabilizing protocol: If we authorize any process to execute the critical section only when it receives a new token (*Fwd-action*), they execute the critical section at most twice without satisfying the safety. We now describe how to build a snap-stabilizing mutual exclusion using $\mathcal{SSBB}(\mathcal{DFS})$. By Theorem 5.1, from the moment when a process becomes requestor, it just has to wait the reception of at least three tokens and then executes the critical section when receiving the third token to satisfy the safety of the specification (the liveness is simply ensured by the fact that

Algorithm 5. $\mathcal{ME}$, $\forall p \in V$:

Variable: $Counter_p \in \{0,1,2\}$;
Actions:

ME_1	::	$(Counter_p = 0)$	$\rightarrow$	$Counter_p := 1$; /* *Starting Action* */
ME_2	::	$(Counter_p = 1)$	$\rightarrow$	$Counter_p := 2$;
ME_3	::	$(Counter_p = 2)$	$\rightarrow$	$\langle CS \rangle$; $Counter_p := 0$;

the token circulation is *perpetual*). That is the solution we propose. Our snap-stabilizing mutual exclusion corresponds to the following composite protocol: $\mathcal{ME} \circ|_{Fwd-Guard(p)} \mathcal{SSBB}(\mathcal{DFS})$ where $Fwd\text{-}Guard(p)$ is the guard of $Fwd\text{-}action$ at p in $\mathcal{SSBB}(\mathcal{DFS})$, that is, $Fwd\text{-}Guard(p) \equiv (Forward(p) \wedge Ok(p))$ and $\mathcal{ME}$ is the protocol given in Algorithm 5 (a version with explicit request is given in the appendix, Subsection 6). $\mathcal{ME} \circ|_{Fwd-Guard(p)} \mathcal{SSBB}(\mathcal{DFS})$ works as follows:

1. Since a process p requests the critical section, it waits until $Count_p = 0$ and then starts the execution of $\mathcal{ME}$ by executing ME_1 while receiving a new token by $Fwd\text{-}action$ (following the composition rules, p always executes the $\mathcal{ME}$ actions in the same steps than $Fwd\text{-}action$). By executing ME_1, p sets $Counter_p$ to 1 to remember that it already received one token.
2. When p receives another token ($Fwd\text{-}action$), it executes ME_2 in the same step. By this action, p simply increments $Counter_p$.
3. Finally, by ME_3, p executes the critical section ($\langle CS \rangle$) and, then resets $Counter_p$ to 0 (meaning that it is ready to receive another request) in the same steps than the third $Fwd\text{-}action$.

By Theorem 5.1, the previous described mechanism guarantees that the safety of the mutual exclusion is always satisfied. Also, the liveness is ensured by the fact that $\mathcal{SSBB}(\mathcal{DFS})$ ensures that any process receives tokens infinitely often even if the daemon is unfair. Hence, we can conclude:

THEOREM 5.2. *$\mathcal{ME} \circ|_{Fwd-Guard(p)} \mathcal{SSBB}(\mathcal{DFS})$ is a snap-stabilizing mutual exclusion protocol assuming a distributed unfair daemon.*

Note that, starting from any configuration, our snap-stabilizing mutual exclusion protocol does not prevent several processes from executing the critical section simultaneously. However, it guarantees that each process which requests the critical section during an execution will execute it alone. So, when several processes execute the critical section simultaneously, none of them have requested it before. On the contrary, since a process becomes requestor, we guarantee that it will enter in the critical section only when the section will be free and when it will be the only process able to enter in; that is, it will be the only token holder.

6. CONCLUSION

In the context of single-initiator wave protocols where decisions only occur at the initiator (a fundamental class of distributed protocols) we show that there exists a basic property called *BreakingIn* which is strictly necessary for stabilizing

protocols. Roughly speaking, this property ensures that the initiator can be involved neither in a deadlock nor in a livelock. A protocol with such a property is easy to automatically snap-stabilize using a snap-stabilizing reset protocol. Now, starting from a classical non-fault-tolerant protocol, we just need to slightly modify it into a version having the *BreakingIn* property—which is much easier than directly designing a self- or snap-stabilizing protocol—to obtain a version that can be turn into a snap-stabilizing protocol using our transformer.

The major advantage of our solution compared to Cournier et al. [2003] is that we can now bound the overhead of our transformer. In consequence and contrary to the previous solution, we can obtain snap-stabilizing protocols working under a distributed unfair daemon, the most general daemon of our model. Also, we can analyze the complexity of the transformed protocol. To show the power of our method, we propose two snap-stabilizing applications designed with our transformer: A depth-first token circulation and breath-first spanning tree construction with termination detection. These two protocols work under a distributed unfair daemon and are both efficient in time and space. In particular, our depth-first token circulation constitutes, until now, the best trade-off between the time and the space complexity solving this problem. Also, these two applications show that the requested property does not lead to provide complex codes as that of stabilizing solutions [Huang and Chen 1993; Johnen 1997; Cournier et al. 2005, 2006] but codes are in fact close to basic non-fault-tolerant solutions and as consequence easy to prove.

Also, using a counting property of our transformer, we show how to use our depth-first token circulation to trivially obtain a snap-stabilizing mutual exclusion protocol. Finally, since *BreakingIn* is a property of any self-stabilizing protocol of the same class, we can use our method to snap-stabilize such self-stabilizing protocols Cournier et al. [2006a]. Surprisingly, in addition to snap-stabilization, the method enhances in many cases some other characteristics of the initial self-stabilizing protocol: The time needed for the correct execution (the first execution in the snap-stabilizing version) is drastically reduced and/or the fairness of the daemon can be weakened.

Of course our transformer cannot work with multi-initiator protocols. For example, in a leader election initialized by several processors the partial computations must merge into a single one but, using our transformer, each initiator has initiated a reset. These concurrent resets may mutually cancel the work of the partial computations. Solving this drawback will be the aim of a future work.

APPENDIX

A.1 How to Explicitly Manage the Request?

The external request can be managed into the code of the initiator r using, for instance, a variable $Request_r \in \{Wait, In, Out\}$. $Request_r = Wait$ means that an execution of the protocol is required. When the initialization of the protocol occurs, $Request_r$ switches from *Wait* to *In* meaning that r has taking the request into account. Finally, $Request_r$ switches from *In* to *Out* when the system is

ready to receive another request. Of course, the switching of $Request_r$ from $Wait$ to In and from In to Out is managed by the task itself while the switching from Out to $Wait$ (which means that another execution of the protocol is requested) is externally managed. Note that all other transitions (for instance, In to $Wait$) are forbidden. The switching from Out to $Wait$ can be managed using an external action IR (i.e., *Interface Request*) constituted as follows:

$$\begin{aligned} IR \;::\; & ApplicationRequest(r) \wedge (Request_r = Out) \rightarrow Request_r \\ & := Wait; ApplicationRelease_r; \end{aligned}$$

$ApplicationRequest(r)$ represents the predicate which is true when an application at the initiator r requests an execution. $ApplicationRelease_r$ corresponds to the macro which contains the code of the application that has to be executed when the system takes the request into account. In particular, this macro must make $ApplicationRequest(r)$ false.

The switching from $Wait$ to In has to be performed when the protocol starts. So, the assignment $Request_r := In$ has to be added into the statement of each starting action. Also, a starting action can be executed only if $Request_r = Wait$ (the starting actions are triggered by the requests). So, the condition $Request_r = Wait$ has to be added into the guard of any starting action. Finally, the switching of $Request_r$ from In to Out has to be performed after the termination of the protocol.

A.2 Algorithm $Reset(\mathcal{B})$

$Reset(\mathcal{B})$ is a slightly modified version of $\mathcal{PIF}$. It is divided into three parts: The *PIF*, *question*, and *correction* parts, respectively. The *PIF* part is the most important part of the protocol because it contains the actions related to the three phases of a *PIF* wave: The broadcast phase, the feedback phase following the broadcast phase, and the cleaning phase which cleans the trace of the feedback phase so that the root is ready to broadcast a new message. The two other parts, that is, the question and the correction parts, implement two mechanisms allowing the snap-stabilization of the *PIF* part.

$Reset(\mathcal{B})$ use the four following variables:

1. P_p. This variable points out to the process from which p receives a new broadcast message. So, $P_p \in Neig_p$ if $p \neq r$ and $P_p = \perp$ if $p = r$ (indeed, as r is the initiator of the broadcast, r never receives any message and P_r is a constant). A spanning tree of the network with regard to the P-values is dynamically built during the broadcast phase.
2. L_p. L_p contains the length of the path followed by the broadcast message from r to p. So, L_r is the constant 0 and $\forall p \in V \setminus \{r\}$, p assigns L_p to $L_{P_p} + 1$ each time it receives a new broadcast message. This variable is used to detect the cycles that can form some P variables in the initial configuration. Indeed, in a P cycle, at least, one process q satisfies $L_p \neq L_{P_p} + 1$.
3. PIF_p. Informally, the variable PIF_p allows to know in which phase of the *PIF* the process p is. PIF_p is set to B when p switches to the broadcast phase by $B\text{-}action$. In particular, note that since any *PIF* wave starts by a

Algorithm 6. $Reset(\mathcal{B})$ for $p = r$

Inputs: $Neig_p$: set of (locally) ordered neighbors of p; $\mathcal{B}$: algorithm satisfying $\mathcal{SSBB}$-$Friendly(\mathcal{T},\mathcal{D})$;
Input-Output: $Request_r \in \{Wait,In,Out\}$ (shared with the IR action);
Constants: $P_p = \perp$; $L_p = 0$;
Variables: $PIF_p \in \{B,F,P,C\}$; $Que_p \in \{Q,R,A\}$;
Macro:

$$Child_p = \{q \in Neig_p :: (PIF_q \neq C) \wedge (P_q = p) \wedge (L_q = L_p + 1) \wedge [(PIF_q \neq PIF_p) \Rightarrow (PIF_p \in \{B,P\} \wedge PIF_q = F)]\};$$

Predicates:

$$\begin{array}{lcl}
\boldsymbol{Ok}(p) & \equiv & (PIF_p = C)\\
CFree(p) & \equiv & (\forall q \in Neig_p :: PIF_q \neq C)\\
Leaf(p) & \equiv & [\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (P_q \neq p)]\\
BLeaf(p) & \equiv & (PIF_p = B) \wedge [\forall q \in Neig_p :: (P_q = p) \Rightarrow (PIF_q = F)]\\
AnswerOk(p) & \equiv & (Que_p = A) \wedge [\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Que_q = A)]\\
Broadcast(p) & \equiv & (PIF_p = C) \wedge Leaf(p)\\
Feedback(p) & \equiv & BLeaf(p) \wedge CFree(p) \wedge AnswerOk(p)\\
PreClean(p) & \equiv & (PIF_p = F) \wedge [\forall q \in Neig_p :: (P_q = p) \Rightarrow (PIF_q \in \{F,C\})]\\
Cleaning(p) & \equiv & (PIF_p = P) \wedge Leaf(p)\\
Require(p) & \equiv & (PIF_p \in \{B,F\}) \wedge [(PIF_p = B) \Rightarrow CFree(p)]\\
 & & \wedge [[(Que_p = Q) \wedge (\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Que_q \in \{Q,R\}))]\\
 & & \vee [(Que_p = A) \wedge (\exists q \in Neig_p :: (PIF_q \neq C)\\
 & & \wedge ((Que_q = Q) \vee (q \in Child_p \wedge Que_q = R)))]]\\
Answer(p) & \equiv & (PIF_p \in \{B,F\}) \wedge [(PIF_p = B) \Rightarrow CFree(p)]\\
 & & \wedge (Que_p = R) \wedge (\forall q \in Child_p :: Que_q \in \{W,A\})\\
 & & \wedge [\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Que_q \neq Q)]
\end{array}$$

Actions:

$$\begin{array}{lll}
/^* \ PIF\ Part\ ^*/ & & \\
B\text{-}action & :: (\mathcal{B}.Request_r = Wait) \wedge \mathcal{B}.End(p) \wedge Broadcast(p) & \rightarrow PIF_p := B; Que_p := Q;\\
 & & \quad \mathcal{B}.Request_r := In;\\
F\text{-}action & :: Feedback(p) & \rightarrow PIF_p := F; \mathcal{B}.Init_p;\\
P\text{-}action & :: PreClean(p) & \rightarrow PIF_p := P;\\
C\text{-}action & :: Cleaning(p) & \rightarrow PIF_p := C;\\
T\text{-}action & :: (B.Request_r = In) \wedge B.End(p) \wedge (PIF_p = C) & \rightarrow \mathcal{B}.Request_r := Out;\\
/^* \ Question\ Part\ ^*/ & & \\
QR\text{-}action & :: Require(p) & \rightarrow Que_p := R;\\
QA\text{-}action & :: Answer(p) & \rightarrow Que_p := A;
\end{array}$$

broadcast phase initiated by r, B-*action* at r is the only starting action of PIF. Then, PIF_p is set to F when p switches to the feedback phase by F-*action*. Finally, the cleaning phase is managed with the two states P and C. After r detects the end of the feedback phase, r initiates using P-*action* a broadcast of the P value along the spanning tree computed during the broadcast phase to inform all the processes of this termination. Then, the processes successively switches to C by C-*action* from the leaves to r meaning that they now ready to receive another message. Hence, the PIF wave terminates when r sets PIF_r to C by C-*action*. Finally, note that two other states exist in PIF_p for $p \neq r$: EB and EF. They are used by the correction part only.

Algorithm 7. $Reset(\mathcal{B})$ for $p \neq r$

Inputs: $Neig_p$: set of (locally) ordered neighbors of p; $\mathcal{B}$: algorithm satisfying $\mathcal{SSBB}$-$Friendly(\mathcal{T},\mathcal{D})$;

Variables: $PIF_p \in \{B,F,P,C,EB,EF\}$; $P_p \in Neig_p$; $L_p \in \mathbb{N}$; $Que_p \in \{Q,R,W,A\}$;

Macros:

$Child_p = \{q \in Neig_p :: (PIF_q \neq C) \wedge (P_q = p) \wedge (L_q = L_p + 1) \wedge [(PIF_q \neq PIF_p) \Rightarrow ((PIF_p \in \{B,P\} \wedge PIF_q = F) \vee (PIF_p = EB))]\}$;

$Pre\_Potential_p = \{q \in Neig_p :: PIF_q = B\}$;

$Potential_p = \{q \in Neig_p :: \forall q' \in Pre\_Potential_p, L_q \leq L_{q'}\}$;

Predicates:

$\mathbf{Ok}(p) \equiv (PIF_p = C)$

$CFree(p) \equiv (\forall q \in Neig_p :: PIF_q \neq C)$

$Leaf(p) \equiv [\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (P_q \neq p)]$

$BLeaf(p) \equiv (PIF_p = B) \wedge [\forall q \in Neig_p :: (P_q = p) \Rightarrow (PIF_q = F)]$

$AnswerOk(p) \equiv (Que_p = A) \wedge [\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Que_q = A)]$

$GoodPIF(p) \equiv (PIF_p = C) \vee [(PIF_{P_p} \neq PIF_p) \Rightarrow ((PIF_{P_p} = EB) \vee (PIF_p = F \wedge PIF_{P_p} \in \{B,P\}))]$

$GoodL(p) \equiv (PIF_p \neq C) \Rightarrow (L_p = L_{P_p} + 1)$

$AbRoot(p) \equiv \neg GoodPIF(p) \vee \neg GoodL(p)$

$EFAbRoot(p) \equiv (PIF_p = EF) \wedge AbRoot(p) \wedge [\forall q \in Neig_p :: (P_q = p \wedge L_q > L_p) \Rightarrow (PIF_q \in \{EF,C\})]$

$EBroadcast(p) \equiv (PIF_p \in \{B,F,P\}) \wedge [\neg AbRoot(p) \Rightarrow (PIF_{P_p} = EB)]$

$EFeedback(p) \equiv (PIF_p = EB) \wedge [\forall q \in Neig_p :: (P_q = p \wedge L_q > L_p) \Rightarrow (PIF_q \in \{EF,C\})]$

$Broadcast(p) \equiv (PIF_p = C) \wedge (Potential_p \neq \emptyset) \wedge Leaf(p)$

$Feedback(p) \equiv BLeaf(p) \wedge CFree(p) \wedge AnswerOk(p)$

$PreClean(p) \equiv (PIF_p = F) \wedge (PIF_{P_p} = P) \wedge [\forall q \in Neig_p :: (P_q = p) \Rightarrow (PIF_q \in \{F,C\})]$

$Cleaning(p) \equiv (PIF_p = P) \wedge Leaf(p)$

$Require(p) \equiv (PIF_p \in \{B,F\}) \wedge [(PIF_p = B) \Rightarrow CFree(p)] \wedge [[(Que_p = Q) \wedge (\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Que_q \in \{Q,R\}))] \vee [(Que_p \in \{W,A\}) \wedge (\exists q \in Neig_p :: (PIF_q \neq C) \wedge ((Que_q = Q) \vee (q \in Child_p \wedge Que_q = R)))]]$

$Wait(p) \equiv (PIF_p \in \{B,F\}) \wedge [(PIF_p = B) \Rightarrow CFree(p)] \wedge (Que_p = R) \wedge (Que_{P_p} = R) \wedge (\forall q \in Child_p :: Que_q \in \{W,A\}) \wedge (\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Que_q \neq Q))$

$Answer(p) \equiv (PIF_p \in \{B,F\}) \wedge [(PIF_p = B) \Rightarrow CFree(p)] \wedge (Que_p = W) \wedge (Que_{P_p} = A) \wedge (\forall q \in Child_p :: Que_q \in \{W,A\}) \wedge (\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Que_q \neq Q))$

Actions:

/* Correction Part */

EC-action	::	$EFAbRoot(p)$	$\rightarrow$	$PIF_p := C$;
EB-action	::	$EBroadcast(p)$	$\rightarrow$	$PIF_p := EB$;
EF-action	::	$EFeedback(p)$	$\rightarrow$	$PIF_p := EF$;

/* PIF Part */

B-action	::	$Broadcast(p)$	$\rightarrow$	$PIF_p := B$; $P_p := \min_{\prec_p}(Potential_p)$; $L_p := L_{P_p} + 1$; $Que_p := Q$;
F-action	::	$Feedback(p)$	$\rightarrow$	$PIF_p := F$; $\mathcal{B}.Init_p$;
P-action	::	$PreClean(p)$	$\rightarrow$	$PIF_p := P$;
C-action	::	$Cleaning(p)$	$\rightarrow$	$PIF_p := C$;

/* Question Part */

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

4. Que_p. This variable is used in the question part. Roughly speaking, since a process receives a message broadcast by the root, this part controls that it switches to the feedback phase only after all its neighbors have received the message.

A.3 Explicit Request in $\mathcal{ME}$

We define the variable $Request_p \in \{Wait, In, Out\}$ into the code of any process p to manage the requests for the critical section. Using $Request_p$, $\mathcal{ME} \circ|_{Fwd-Guard(p)} \mathcal{SSBB}(\mathcal{DFS})$ works as follows:

1. When a process p requests the critical section (*i.e.*, $Request_p = Wait$), it waits the reception of a new token by *Fwd-action*. When it executes *Fwd-action*, p also executes ME_1 in the same step. By executing ME_1, p initializes $Counter_p$ to 1 (to remember that it already received one token) and switches $Request_p$ to In (meaning that the request has been taken into account).
2. When p receives another token (*Fwd-action*), it satisfies $Request_p = In \wedge Counter_p = 1$. It then executes ME_2 in the same step than *Fwd-action*. By this action, p simply increments $Counter_p$.
3. Finally, by ME_3, p executes the critical section ($< CS >$) and, then, switches $Request_p$ to Out (meaning that the requested critical section has been executed by p) in the same steps than the third *Fwd-action*.

Algorithm 8. $\mathcal{ME}$, $\forall p \in V$:

Variables: $Request_p \in \{Wait, In, Out\}$; $Counter_p \in \{1,2\}$;

Actions:

ME_1	::	$(Request_p = Wait)$	$\rightarrow$	$Counter_p := 1; Request_p := In;$
ME_2	::	$(Request_p = In) \wedge (Counter_p = 1)$	$\rightarrow$	$Counter_p := 2;$
ME_3	::	$(Request_p = In) \wedge (Counter_p = 2)$	$\rightarrow$	$\langle CS \rangle; Request_p := Out;$

REFERENCES

AWERBUCH, B. AND GALLAGER, R. G. 1985. Distributed bfs algorithms. In *Proceedings of the Annual Symosium on Foundations of Computer Science*. IEEE, 250–256.

BUI, A., DATTA, A., PETIT, F., AND VILLAIN, V. 1999. State-optimal snap-stabilizing PIF in tree networks. In *Proceedings of the 4th Workshop on Self-Stabilizing Systems*. IEEE Computer Society Press, 78–85.

COURNIER, A., DATTA, A., PETIT, F., AND VILLAIN, V. 2003. Enabling snap-stabilization. In *Proceedings of the 23th International Conference on Distributed Computing Systems (ICDCS'03)*. IEEE Computer Society Press, 12–19.

COURNIER, A., DEVISMES, S., PETIT, F., AND VILLAIN, V. 2004. Snap-stabilizing depth-first search on arbitrary networks. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS'04)*. Lecture Notes in Computer Science, vol. 3544, 267–282.

COURNIER, A., DEVISMES, S., PETIT, F., AND VILLAIN, V. 2006. Snap-stabilizing depth-first search on arbitrary networks. *Comput. J. 49,* 3, 268–280.

COURNIER, A., DEVISMES, S., AND VILLAIN, V. 2005. A snap-stabilizing DFS with a lower space requirement. In *Proceedings of the 7th International Symposium on Self-Stabilizing Systems (SSS'05)*. Lecture Notes in Computer Science, vol. 3764, 33–47.

COURNIER, A., DEVISMES, S., AND VILLAIN, V. 2006a. From self- to snap- stabilization. In *Proceedings of the 8th International Symposium on Self-Stabilization, Safety, and Security (SSS'06)*. Lecture Notes in Computer Science, vol. 4280, 199–213.

COURNIER, A., DEVISMES, S., AND VILLAIN, V. 2006b. Snap-stabilizing PIF and useless computations. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems (ICPADS'06)*. Vol. 1. IEEE Computer Society Press, 39–46.

COURNIER, A., DEVISMES, S., AND VILLAIN, V. 2007. Light enabling snap-stabilization. Tech. rep. 2007-04, LaRIA, CNRS FRE 2733.

DELAËT, S., DUCOURTHIAL, B., AND TIXEUIL, S. 2005. Self-stabilization with r-operators revisited. In *Self-Stabilizing Systems*, T. Herman and S. Tixeuil, Eds. Lecture Notes in Computer Science, vol. 3764, 68–80.

DIJKSTRA, E. 1974. Self stabilizing systems in spite of distributed control. *Comm. ACM 17*, 643–644.

DOLEV, S., ISRAELI, A., AND MORAN, S. 1997. Uniform dynamic self-stabilizing leader election. *IEEE Trans. Parall. Distrib. Syst. 8,* 4, 424–440.

DOLEV, S. AND TZACHAR, N. 2006. Empire of colonies: Self-stabilizing and self-organizing distributed algorithms. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*. Number 4305 in Lecture Notes in Computer Science. Springer, 230–243.

HUANG, S. AND CHEN, N. 1993. Self-stabilizing depth-first token circulation on networks. *Distrib. Comput. 7*, 61–66.

HUANG, S.-T. AND CHEN, N.-S. 1992. A self-stabilizing algorithm for constructing breadth-first trees. *Inf. Process. Lett. 41,* 2, 109–117.

JOHNEN, C. 1997. Memory efficient, self-stabilizing algorithm to construct bfs spanning trees. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC'97)*. ACM Press, New York, NY, 288.

JOHNEN, C., ALIMA, L., DATTA, A. K., AND TIXEUIL, S. 2002. Optimal snap-stabilizing neighborhood synchronizer in tree networks. *Parall. Proc. Lett. 12,* 3-4, 327–340.

KATZ, S. AND PERRY, K. 1993. Self-stabilizing extensions for message-passing systems. *Distrib. Comput. 7*, 17–26.

TEL, G. Second edition 2001. *Introduction to Distributed Algorithms*. Cambridge University Press, Cambridge, UK.

VILLAIN, V. 2002. Snap-stabilization versus self-stabilization. *Proceedings of the Journées Internationales sur l'auto-Stabilisation.* CIRM, Luminy France.

Received February 2007; revised July 2008; accepted September 2008