An Improved Snap-Stabilizing PIF Algorithm

Lélia Blin, Alain Cournier, and Vincent Villain

LaRIA, Université de Picardie Jules Verne, France
5, rue du Moulin Neuf, 80000 Amiens, France
{blin,cournier,villain}@laria.u-picardie.fr

Abstract. A *snap-stabilizing protocol*, starting from any arbitrary initial configuration, always behaves according to its specification. In [10], Cournier and al. present the first snap-stabilizing Propagation of Information with Feedback (PIF) protocol in arbitrary networks. But, in order to achieve the desirable property of snap-stabilization, the algorithm needs the knowledge of the exact size of the network. This drawback prevents the protocol from working on dynamical systems. In this paper, we propose an original protocol which solves this drawback.

Keywords: Fault-tolerance, propagation of information with feedback, reset protocols, self-stabilization, snap-stabilization, wave algorithms.

1 Introduction

Chang [8] and Segall [18] defined the concept of *Propagation of Information with Feedback* (PIF) (also called *wave propagation*). A processor p initiates the first phase of the wave: the propagation or broadcast phase. Every processor, upon receiving the first broadcast message, chooses the sender of this message as its parent in the PIF wave, and forwards the wave to its neighbors except its parent. When a processor receives a feedback (acknowledgment) message from all its children with respect to the current PIF wave, it sends a feedback message to its parent. So, eventually, the feedback phase ends at p. Broadcast with feedback scheme has been used extensively in distributed computing to solve a wide class of problems, e.g., spanning tree construction, distributed infimum function computations, snapshot, termination detection, and synchronization (see [17, 19, 16] for details). So, designing efficient fault-tolerant wave algorithms is an important task in the distributed computing research.

The concept of *self-stabilization* [12] is the most general technique to design a system to tolerate arbitrary transient faults. A self-stabilizing system, regardless of the initial states of the processors and initial messages in the links, is guaranteed to converge to the intended behavior in finite time. *Snap-stabilization* was introduced in [7]. A *snap-stabilizing* algorithm guarantees that it always behaves according to its specification. In other words, a snap-stabilizing algorithm is also a self-stabilizing algorithm which stabilizes in 0 steps. Obviously, a *snap-stabilizing* protocol is optimal in stabilization time.

Related Work. PIF algorithms have been proposed in the area of self-stabilization, e.g., [7, 13, 15] for tree networks, and [9, 10, 20] for arbitrary networks. The self-stabilizing PIF protocols have also been used in the area of self-stabilizing synchronizers [2, 4, 6]. The most general method to "repair" the system is to reset the entire system after a transient fault is detected. Reset protocols are also PIF-based algorithms. Several reset protocols exist in the self-stabilizing literature (see [1, 3, 4, 5, 20]). Self-stabilizing snapshot algorithms [14, 20] are also based on the PIF scheme. The first snap-stabilizing PIF algorithm for arbitrary networks has been presented in [10].

Contribution. In [10], the system needs the knowledge of the exact size of the network (i.e., the number of processors). So this size must be constant and the protocol cannot work on dynamical networks. In this paper, we solve this drawback by the composition of three protocols. The first one is the actual PIF protocol, the second one allows the processors to execute the feedback phase, and the role of the third one is to deal with the processors having an abnormal state.

Outline of the paper. In the next section (Section 2), we describe the distributed system and the model in which our PIF scheme is written. In the same section, we also state what it means for a protocol to be snap-stabilizing and give a formal statement of the problem solved in this paper. The PIF algorithm is presented in Section 3. We then prove the correctness of the algorithm in Section 4, followed by the complexity analysis. Finally, we make some concluding remarks in Section 5.

2 Preliminaries

Distributed System. We consider an asynchronous network of N processors connected by bidirectional communication links according to an arbitrary topology. We consider networks which are *asynchronous*. $Neig_p$ denotes the set of neighbors of processor p ($Neig_p$ is shown as an input from the system). We consider the local shared memory model of communication. The program of every processor consists of a set of *shared variables* (henceforth, referred to as variables) and a finite set of actions. A processor can only write to its own variables, and read its own variables and variables owned by the neighboring processors.

Each action is of the following form: $<$ *label* $>$::$<$ *guard* $>\longrightarrow<$ *statement* $>$. The guard of an action in the program of p is a boolean expression involving the variables of p and its neighbors. The statement of an action of p updates one or more variables of p. An action can be executed only if its guard evaluates to true. We assume that the actions are atomically executed, meaning, the evaluation of a guard and the execution of the corresponding statement of an action, if executed, are done in one atomic step.

The *state* of a processor is defined by the value of its variables. The *state* of a system is the product of the states of all processors ($\in V$). We will refer to the state of a processor and system as a (*local*) *state* and (*global*) *configuration*, respectively. Let a distributed protocol $\mathcal{P}$ be a collection of binary transition relations denoted by $\mapsto$, on $\mathcal{C}$, the set of all possible configurations of the system. A *computation* of a protocol $\mathcal{P}$ is a *maximal* sequence of configurations $e = \gamma_0, \gamma_1, \ldots, \gamma_i, \gamma_{i+1}, \ldots$, such that for $i \geq 0, \gamma_i \mapsto \gamma_{i+1}$ (a single *computation step*) if γ_{i+1} exists, or γ_i is a terminal configuration. *Maximality* means that the sequence is either infinite, or it is finite and no action of $\mathcal{P}$ is enabled in the final configuration. All computations considered in this paper are assumed to be maximal. The set of all possible computations of $\mathcal{P}$ in system S is denoted as $\mathcal{E}$. A processor p is said to be *enabled* in γ ($\gamma \in \mathcal{C}$) if there exists an action A such that the guard of A is true in γ. We consider that any processor p executed a *disable action* in the computation step $\gamma_i \mapsto \gamma_{i+1}$ if p was enabled in γ_i and not enabled in γ_{i+1}, but did not execute any action between these two configurations. (The disable action represents the following situation: At least one neighbor of p changed its state between γ_i and γ_{i+1}, and this change effectively made the guard of all actions of p false.) Similarly, an action A is said to be enabled (in γ) at p if the guard of A is true at p (in γ).

We assume a *weakly fair and distributed daemon.* The *weak fairness* means that if a processor p is continuously enabled, then p will be eventually chosen by the daemon to execute an action. The *distributed* daemon implies that during a computation step, if one or more processors are enabled, then the daemon chooses at least one (possibly more) of these enabled processors to execute an action.

In order to make our algorithm more readable, we designed it as a *composition* of three algorithms. In this composition, if a processor p is enabled for k of the combined protocols, then, if the daemon chooses it, p executes an enabled action of each of the k protocols, in the same step. Variables, predicates, or macros of an algorithm A used by an algorithm B are shown as inputs in Algorithm B.

In order to compute the time complexity measure, we use the definition of *round* [13]. This definition captures the execution rate of the slowest processor in any computation. Given a computation e ($e \in \mathcal{E}$), the *first round* of e (let us call it e') is the minimal prefix of e containing the execution of one action (an action of the protocol or the disable action) of every continuously enabled processor from the first configuration. Let e'' be the suffix of e, i.e., $e = e'e''$. The *second round* of e is the first round of e'', and so on.

Snap-Stabilization. Let $\mathcal{X}$ be a set. $x \vdash P$ means that an element $x \in \mathcal{X}$ satisfies the predicate P defined on the set $\mathcal{X}$.

Definition 1 (Snap-Stabilization). *Let $\mathcal{T}$ be a task, and $\mathcal{SP}_\mathcal{T}$ a specification of $\mathcal{T}$. The protocol $\mathcal{P}$ is snap-stabilizing for the specification $\mathcal{SP}_\mathcal{T}$ on $\mathcal{E}$ if and only if the following condition holds: $\forall e \in \mathcal{E} :: \; e \vdash \mathcal{SP}_\mathcal{T}$.*

The Problem To Be Solved. Any processor can be an initiator in a PIF protocol, and several PIF protocols may run simultaneously. We consider the problem in this paper in a general setting of the PIF scheme where we assume that the PIF is initiated by a processor, called the *root.* We denote the root processor by r.

Specification 1 (PIF Cycle) *A finite computation $e = \gamma_0, \ldots, \gamma_i, \gamma_{i+1}, \ldots, \gamma_t \in \mathcal{E}$ is called a PIF Cycle, if and only if the following condition is true: If Processor 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 processor $p \neq r$.*

Remark 1. So 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: 1. if r has a message m to broadcast, then it will be able to start the broadcast in a finite time, and 2. starting from any configuration where r is ready to broadcast, the system satisfies Specification 1.

3 Algorithm

The snap-stabilizing PIF algorithm we proposed is divided in three parts: PIF Algorithm (Algorithms 1 and 2), Question Algorithm (Algorithms 3 and 4), and Error Algorithm (Algorithms 5 and 6). PIF Algorithm is the main algorithm. It works in three phases: 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. Question Algorithm controls that the processors do not execute the feedback phase too early. Error Algorithm cleans the processors which do not have a normal configuration.

We first present the normal behavior of the PIF algorithm. We then explain the method of error correction.

3.1 Normal Behavior

Consider the configuration where $\forall p, Pif_p = C$. We refer to this configuration as the *normal starting configuration*. In this configuration, the root is the only enabled processor. The root broadcasts a message m and switches to the broadcast phase by executing $Pif_r = B$ (*B-action*). When a processor p (such that $Pif_p = C$) waiting for a message finds one of its neighbors q in the broadcast phase, p receives the message from q. Then, p sets its variable Pif_p to B, points to q using the variable Par_p, and sets its level L_p to $L_q + 1$ (*B-action*). Typically, L_p contains the length of the path followed by the broadcast message from the root r to p. (Since r never receives a broadcast message from any of its neighbor, r does not have any variable Par and L_r is shown as a **constant** in the algorithm.) Processor p is now in the broadcast phase ($Pif_p = B$) and is supposed to broadcast the message to its neighbors (except Par_p). So, step by step, a spanning tree (w.r.t. the variable Par) rooted at r is dynamically built during the broadcast phase. Let us call this tree the *B-tree$_r$*. Each time a processor broadcast m, it also executes $Que_p := Q$ (*QB-action* in Question

Algorithm). When its neighbors (with variable $Pif \neq C$) have taken p's question in account by setting Que to R (*QR-action*), p can also execute *QR-action*, meaning that it send a request to r: "Do you authorize me to feedback?". Eventually, some processors in $B\text{-}tree_r$ cannot broadcast the message because all its neighbors have received the message from some other neighbor. The processors which are not able to broadcast the message further are the leaves of $B\text{-}tree_r$. In this case the leaves execute $Que_p := W$ (*QW-action*), meaning that now they are waiting for an answer from r. This action is propagated toward the root if possible. (p propagates it if all its children in $B\text{-}tree_r$ have setted their Que variable to W and no neighbor has still $Pif_p = C$.) With similar conditions, r executes its *QA-action*: it sends an answer to its children (*QA-action*) meaning that they are authorized to feedback. When a leaf receives this answer, it can execute the feedback phase (*F-action*). So, step by step, every processor p propagates the feedback phase towards the root in $B\text{-}tree_r$ by executing *F-action*. The feedback phase eventually reaches the root r. Finally, the leaf processors in $B\text{-}tree_r$ initiate the third phase, called the *cleaning phase*. The aim of this phase is to erase the trace of the last PIF cycle (the broadcast phase followed by the feedback phase) initiated by the root, i.e., to bring the system in the normal starting configuration again ($\forall p, Pif_p = C$). A leaf processor p in $B\text{-}tree_r$ initiates the cleaning phase by setting Pif_p to C when each of its neighbors q is either in the feedback phase ($Pif_q = F$) or in the cleaning phase ($Pif_q = C$). So, the cleaning phase works in parallel and follows the feedback phase. Once all neighbors of the root change to the cleaning phase, the root also participates in the cleaning phase.

3.2 Error Correction

During the normal behavior, the processors must maintain some properties based on the value of their variables and that of their parent. For the processors p which are not the root ($p \neq r$), we list some of those conditions below:

1. If p is in the broadcast phase, then its parent is also in the broadcast phase. Also, if p is in the feedback phase, then its parent is either in the broadcast or feedback phase (Predicate *GoodPif* in PIF Algorithm).
2. If p is involved in the PIF Cycle ($Pif_p \neq C$), then its level L_p must be equal to one plus the level of its parent (Predicate *GoodLevel* in PIF Algorithm).
3. If p is involved in the PIF Cycle it must satisfy (1) and (2) (Predicate $\neg$*AbRoot* in PIF Algorithm).

Starting now from any configuration, a processor p may satisfy *AbRoot*. In this case, we cannot simply set Pif_p to C. Assume that $Pif_p = B$. Since p satisfies *AbRoot*, that means that some processors in the broadcast phase can be in the abnormal tree rooted in p ($B\text{-}tree_p$). If we simply set Pif_p to C, p can participate again to the broadcast of the tree of which it was the root. Since we do not assume the knowledge of any bound on the L values (we may assume that the maximum value of L is any upper bound of N), this scheme can progress infinitely often (respectively, too many times), and the system contains an abnormal tree which can prevent (respectively, dramaticaly slow down) the progression of the tree of the normal broadcast phase (*B-tree*$_r$). Error Algorithm solves this problem by paralyzing the progress of any abnormal tree before to remove it. A processor p can broadcast a message from a neighbor q only if q satisfies $Pif_q = B$ and $FreeError(q)$, i.e., $E_q = C$ (see *Potential* and *Pre_Potential* in PIF Algorithm). So, if p is an abnormal root, it sets its variable E_p to B and broadcasts this value in its tree (and only in its tree). When p receives an aknowledgment of all its children (value F of variable E), it knows that all the processors q of its tree have $E_q = F$ and no processor can now participate in the broadcast of q. Then p can leave its tree and it will not try to broadcast a message of one of the processors q before q broadcasts a message of another tree. By this process, all abnormal trees eventually disappear, and *B-tree*$_r$ will be able to grow until it reaches all the processors of the network.

Question Algorithm has now to deal with abnormal trees meaning that some processor p broadcasting the message from r can execute the *B-action* while q, one of its neighbors, belongs to an abnormal tree. Then, to prevent q to set Que_q to A, p is waiting for q to set Que_q to R (*QR-action*). This value will erase all A values in the path from q to the abnormal root. Since only r can generate a A value, q will never receives any A and will eventually leave the abnormal tree as described above.

Algorithm 1 $\mathcal{PIF}$ for the root ($p = r$).

```
Input:
    $Neig_p$: set of (locally) ordered neighbors of $p$
    $AnswerOK()$: predicate from Question Algorithm
Constant:
    $L_p = 0$
Variables:
    $Pif_p \in \{B, F, C\}$
Predicates:
    $Leaf(p) \equiv (\forall q \in Neig_p :: (Pif_q \neq C) \Rightarrow (Par_q \neq p))$;
    $Broadcast(p) \equiv (Pif_p = C) \wedge Leaf(p)$;
    $CFree(p) \equiv (\forall q \in Neig_p :: (Pif_q \neq C)$;
    $BLeaf(p) \equiv (Pif_p = B) \wedge (\forall q \in Neig_p :: (Par_q = p) \Rightarrow (Pif_q = F))$;
    $Feedback(p) \equiv BLeaf(p) \wedge CFree(p) \wedge AnswerOK(p)$;
    $Cleaning(p) \equiv (Pif_p = F) \wedge Leaf(p) \wedge (\forall q \in Neig_p :: Pif_q \neq B))$;
Actions:
B-action :: $Broadcast(p) \rightarrow Pif_p := B$;
F-action :: $Feedback(p) \rightarrow Pif_p := F$
C-action :: $Cleaning(p) \rightarrow Pif_p := C$
```

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

```
Input:
    $Neig_p$: set of (locally) ordered neighbors of $p$
    $AnswerOK()$: predicate from Question Algorithm
    $FreeError()$, $CError()$: predicates from Error Algorithm
Variables:
    $Pif_p \in \{B, F, C\}$
    $L_p : integer$
    $Par_p \in Neig_p$
Macros:
    $Pre\_Potential_p = \{q \in Neig_p :: (Pif_q = B) \wedge (Par_q \neq p) \wedge FreeError(q)\};$
    $Potential_p = \{q \in Pre\_Potential :: \forall q' \in Pre\_Potential, L_q \leq L_{q'}\};$
    $Child_p \equiv \{q \in Neig_p :: (Pif_q \in \{B, F\}) \wedge (Par_q = p)$
        $\wedge[(Pif_p \neq Pif_q) \Rightarrow (Pif_p = B)] \wedge (L_q = L_p + 1)\};$
Predicates:
    $Leaf(p) \equiv (\forall q \in Neig_p :: (Pif_q \neq C) \Rightarrow (Par_q \neq p));$
    $Broadcast(p) \equiv (Pif_p = C) \wedge Leaf(p) \wedge (Potential_p \neq \emptyset);$
    $CFree(p) \equiv (\forall q \in Neig_p :: (Pif_q \neq C);$
    $BF(p) \equiv (Pif_p = B) \vee (Pif_p = F);$
    $GoodPif(p) \equiv BF(p) \Rightarrow ((Pif_{Par_p} \neq Pif_p) \Rightarrow (Pif_{Par_p} = B));$
    $GoodLevel(p) \equiv BF(p) \Rightarrow (L_p = L_{Par_p} + 1);$
    $AbRoot(p) \equiv \neg GoodPif(p) \vee \neg GoodLevel(p);$
    $Normal(p) \equiv \neg AbRoot(p) \wedge FreeError(p);$
    $BLeaf(p) \equiv (Pif_p = B) \wedge (\forall q \in Neig_p :: (Par_q = p) \Rightarrow (Pif_q = F));$
    $Feedback(p) \equiv BLeaf(p) \wedge Normal(p) \wedge CFree(p) \wedge AnswerOK(p);$
    $Cleaning(p) \equiv (Pif_p = F) \wedge Normal(p) \wedge Leaf(p) \wedge (\forall q \in Neig_p :: Pif_q \neq B));$
Actions:
    $B$-$action :: Broadcast(p) \rightarrow Pif_p := B; Par_p := min_{\prec_p}(Potential_p); L_p := L_{Par_p} + 1;$
    $F$-$action :: Feedback(p) \rightarrow Pif_p := F$
    $C$-$action :: Cleaning(p) \rightarrow Pif_p := C$
    $E$-$action :: CError(p) \rightarrow Pif_p := C$
```

4 Proof of Correctness

As the system can start in an arbitrary (including an undesirable) configuration, we need to show that the algorithm can deal with all the possible errors. To characterize these erroneous configurations, in Subsection 4.1, we define some terms to distinguish these configurations. Moreover, we must show that despite these erroneous configurations, the system always behaves according to its specifications, i.e., is snap-stabilizing.

4.1 Some Definitions

Definition 2 (E-Trace). *Let Y be a t-uple of processors ($Y = (p_0, p_1, \ldots, p_k)$). $E - trace(Y) = E_0E_1 \ldots E_k$ is the sequence of the values of Variable E on processors p_i ($i = 0 \ldots k$).*

Algorithm 3 Question Algorithm for the root ($p = r$).

```
Input:
    $Neig_p$: set of (locally) ordered neighbors of $p$
    $Par_q$, $Pif_p$, $Pif_q$: variables from $\mathcal{PIF}$
    $CFree(), Broadcast()$: predicate from $\mathcal{PIF}$
Variables:    $Que_p \in \{Q, R, A\}$
[the protocol below concerns only $p$ such that $Pif_p \in \{B, F\}$]
Predicates:
    $Require(p) \equiv ([(Que_p = Q) \wedge (\forall q \in Neig_p :: (Que_q \in \{Q, R\})]$
                  $\vee[(Que_q \in \{W, A\})$
                   $\wedge(\exists q \in Neig_p :: (Que_q = Q) \vee ((Par_q = p) \wedge (Que_q = R)))]);$
    $Answer(p) \equiv (Que_p = R) \wedge CFree(p)$
                  $\wedge(\forall q \in Neig_p :: (Pif_q \neq C) \Rightarrow [((Par_q = p) \wedge (Que_q = W))$
                                                    $\vee((Par_q \neq p) \wedge (Que_q \in \{W, A\}))]);$
 $AnswerOK(p) \equiv (Que_p = A) \wedge (\forall q \in Neig_p :: (Pif_p \neq C) \Rightarrow (Que_q = A));$
Actions:
 $QB\text{-}action :: Broadcast(p) \rightarrow Que_p := Q$
 $QR\text{-}action :: Require(p) \rightarrow Que_p := R$
 $QA\text{-}action :: Answer(p) \rightarrow Que_p := A$
```

Example: Let p and q be two processors such that $E_p = C$ and $E_q = B$. Then $E - Trace(p, q) = CB$.

Definition 3 (E-Prohibited Pairs). *Let p and q such that $Pif_p \neq C$ and $Pif_q \neq C$, and $Par_q = p$. The $E - trace(p, q)$ of type $\{CB, CF, FC, FB\}$ are called E-prohibited pairs. In this case, we also say that $E - trace(p, q)$ is E-prohibited.*

Remark 2. Let p and q such that $Pif_p \neq C$ and $Pif_q \neq C$, and $Par_q = p$. $E - trace(p, q)$ is E-prohibited if and only if $(FCorrection(p) \wedge E_q \neq F) \vee BCorrection(q) \vee (FCorrection(q) \wedge E_p = C)$.

Definition 4 (Path). *The sequence of processors $p_0, p_1, p_2, \ldots p_k$ is called a* path *if $\forall i,\ 1 \leq i \leq k,\ p_i \in Neig_{i-1}$. The path is referred to as an* elementary path *if $\forall i, j, 0 \leq i < j \leq k,\ p_i \neq p_j$. The processors p_0 and p_k are termed as the* extremities *of the path.*

Definition 5 (ParentPath). *For any processor p such that $BF(p)$, a unique path $p_0, p_1, p_2, \ldots p_k = p$, called $ParentPath(p)$, exists if and only if the following conditions are true:* 1. *$\forall i,\ 1 \leq i \leq k,\ Par_{p_i} = p_{i-1}$.* 2. *$\forall i,\ 1 \leq i \leq k,\ BF(p_i) \wedge \neg AbRoot(p_i) \wedge (E - Trace(p_{i-1}, p)$ is not E-prohibited).* 3. *$p_0 = r$ or $AbRoot(p_0)$ or $E - Trace(Par_{p_0}, p_0)$ is E-prohibited.*

Definition 6 (Tree). *For any processor p such that $p = r$ or p is an abnormal processor, we define a set $Tree(p)$ of processors as follows: For any processor q, $q \in Tree(p)$ if and only if p is the first extremity of $ParentPath(q)$.*

Algorithm 4 Question Algorithm for $p \neq r$.

```
Input:
    $Neig_p$: set of (locally) ordered neighbors of $p$
    $Par_p$, $Par_q$, $Pif_p$, $Pif_q$: variables from $\mathcal{PIF}$
    $CFree(), Broadcast()$: predicate from $\mathcal{PIF}$
Variables:    $Que_p \in \{Q, R, W, A\}$
the protocol below concerns only $p$ such that $Pif_p \in \{B, F\}$
Predicates:
    $Require(p) \equiv \neg AbRoot(p)$
                  $\wedge([(Que_p = Q) \wedge (\forall q \in Neig_p :: (Que_q \in \{Q, R\})]$
                    $\vee[(Que_q \in \{W, A\}) \wedge (\exists q \in Neig_p :: (Que_q = Q)$
                                   $\vee((Par_q = p) \wedge (Que_q = R)))]);$
    $Wait(p) \equiv \neg AbRoot(p) \wedge (Que_p = R) \wedge (Que_{Par_p} = R) \wedge CFree(p)$
                  $\wedge(\forall q \in Neig_p :: (Que_q \neq Q) \wedge ((Par_q = p) \Rightarrow (Que_q = W)));$
    $Answer(p) \equiv \neg AbRoot(p) \wedge (Que_p = W) \wedge (Que_{Par_p} = A)$
                  $\wedge(\forall q \in Neig_p :: (Pif_q \neq C) \Rightarrow [((Par_q = p) \wedge (Que_q = W))$
                                   $\vee((Par_q \neq p) \wedge (Que_q \in \{W, A\}))]);$
$AnswerOK(p) \equiv (Que_p = A) \wedge (\forall q \in Neig_p :: (Pif_p \neq C) \Rightarrow (Que_q = A));$
Actions:
 $QB$-action :: $Broadcast(p) \rightarrow Que_p := Q$
 $QR$-action :: $Require(p)$   $\rightarrow Que_p := R$
 $QW$-action :: $Wait(p)$      $\rightarrow Que_p := W$
 $QA$-action :: $Answer(p)$    $\rightarrow Que_p := A$
```

Algorithm 5 Error Algorithm for the root ($p = r$).

```
Constant:
    $E_p = C$
```

Definition 7 (NormalTree). *A tree containing only processors p such that $Normal(p) \vee p = r$ is called a $NormalTree$. Obviously, the system contains only one $NormalTree$: this tree is rooted by r. A tree rooted by another processor than r is called an AbnormalTree.*

Definition 8 (Alive). *A tree T satisfies $Alive(T)$ (or is called Alive) if and only if: $\exists p \in T$ such that $(Pif_p = B) \wedge (E_p = C)$.*

Definition 9 (Dead). *A tree T satisfies $Dead(T)$ (or is called Dead) if and only if$\neg Alive(T)$.*

Remark 3. No processor can hook a Dead tree.

Definition 10 (Alive AbNormal Root: AAR). *A processor p is called AAR if and only if $AbRoot(p) \wedge Alive(Tree(p))$.*

Definition 11 (Falldown Alive AbNormal Root: FAAR). *A processor p is called $FAAR$ if and only if $CError(p) \wedge AAR(p)$.*

Algorithm 6 Error Algorithm for $p \neq r$.

```
Input:
    $Par_p$, $Pif_p$: variables from $\mathcal{PIF}$
    $Child_p$: macro from $\mathcal{PIF}$
    $AbRoot()$, $BF()$: predicates from $\mathcal{PIF}$
Variables:
    $E_p \in \{B, F, C\}$
General Predicate:
$FreeError(p) \equiv BF(p) \Rightarrow (E_p = C)$;
the action below concerns only p such that $Pif_p = C$
Action:
$EB\text{-}init :: Broadcast(p) \to \ E_p := C$

the protocol below concerns only p such that $Pif_p \in \{B, F\}$
Predicates:
      $BError(p) \equiv (E_p = C) \wedge (AbRoot(p) \vee (E_{Par_p} = B)) \wedge (\forall q \in Child_p :: E_q = C)$;
      $FError(p) \equiv (E_p = B) \wedge (AbRoot(p) \vee (E_{Par_p} = B)) \wedge (\forall q \in Child_p :: E_q = F)$;
    $FAbRoot(p) \equiv (E_p = F) \wedge AbRoot(p)$;
 $BCorrection(p) \equiv [(E_p = B) \wedge (\neg AbRoot(p)) \wedge (E_{par_p} \neq B)]$;
 $FCorrection(p) \equiv (E_p = F) \wedge [(\neg AbRoot(p) \wedge E_{par_p} = C) \vee (\exists q \in Child_p :: E_q \neq F)]$;
      $CError(p) \equiv FAbRoot(p) \vee BCorrection(p) \vee FCorrection(p)$;
Actions:
$EB\text{-}action :: BError(p) \to \ E_p := B$
$EF\text{-}action :: FError(p) \to \ E_p := F$
$EC\text{-}action :: CError(p) \to \ E_p := C$
```

Definition 12 (Potential Falldown Alive AbNormal Root:PFAAR). *A processor p is called $PFAAR$ in γ_0 if and only if $\exists e : \gamma_0\gamma_1 \dots \gamma_i \dots$ such that $\exists i \geq 0$, p is $FAAR$ in γ_i.*

4.2 Abnormal Processors

In this subsection, we show that the network contains no abnormal processor in at most $3N - 2$ rounds. We first deduce from the algorithm and Remark 2 the following lemma.

Lemma 1. *Any p in the $NormalTree$ satisfies $E - Trace(ParentPath(p)) \in C^+$, and any p in an $AbnormalTree$ satisfies $E - Trace(ParentPath(p)) \in B^*C^* \cup B^*F^*$.*

Lemma 2. *Error Algorithm never generates E-prohibited pairs.*

Proof. Let p and q such that $Par_q = p$. By checking all the non E-prohibited pairs ($Pif_p = C$, $Pif_q = C$, or $E - trace(p, q) \in \{BB, BC, BF, CC, FF\}$) and the actions of Error Algorithm, it is easy to see that we cannot create any E-prohibited pair.

By checking the actions of PIF and Error Algorithms and by Lemma 2, we can deduce the following result.

Lemma 3. *After the first round the system cannot contain any E-prohibited pair.*

Proof. Let p and q be two processors such that $E-trace(p,q)$ is E-prohibited at the first configuration ($Par_q = p$, $Pif_p \neq C$, $Pif_q \neq C$, and $E-trace(p,q) \in \{CB, CF, FC, FB\}$).

1. Assume that $E-Trace(p,q) = CB$. The only actions of $\mathcal{PIF}$ or Error Algorithm q can execute are *E-action* and *EC-action*, respectively, since $BCorrection(q) \Rightarrow CError(q)$. Until q moves, no action of p can change the value of E_p or set Pif_p to C. So q is continuously enabled and, since the daemon is weakly fair, q will execute the *E-action* and *EC-action* during the first round. After this move, $E-Trace(p,q)$ is never more an E-prohibited pair.
2. Assume that $E-Trace(p,q) = CF$. The case is similar to the previous one.
3. Assume that $E-Trace(p,q) = FC$. The only actions of $\mathcal{PIF}$ or Error Algorithm p can execute are *E-action* and *EC-action*, respectively, since $FCorrection(p) \Rightarrow CError(p)$. If q is not enabled or does not move before p, p will execute *E-action* and *EC-action* during the first round. After this move, $E-Trace(p,q)$ is never more an E-prohibited pair. Assume now that q is enabled and moves before p. Depending on Pif_p and Pif_q, q can satisfy $Feedback(q)$ or $Cleaning(q)$.
   (a) If q satisfies $Feedback(q)$ (in this case $Pif_p = Pif_q = B$), it executes Now q cannot be enabled for $\mathcal{PIF}$ until p moves. Since p is still enabled, the system is still executing the first round, and , as previously, p eventually executes execute the *E-action* and *EC-action* during the first round. After this move, $E-Trace(p,q)$ is never more an E-prohibited pair.
   (b) If q satisfies $Cleaning(q)$ (in this case $Pif_p = Pif_q = F$), executes *C-action*. Now $Pif_q = C$ and $E-Trace(p,q)$ is never more an E-prohibited pair.
4. Assume that $E-Trace(p,q) = FB$. Both p and q satisfy $CError()$. When one of them (or both of them) executes the *E-action* and *EC-action*, $E-Trace(p,q)$ is never more an E-prohibited pair.

By Lemma 2, Error Algorithm never generates E-prohibited pairs. Since we just showed that the initial E-prohibited pairs disappear in one round, the lemma is proved.

Corollary 1. *After the first round, the predicates $BCorrection(p)$ and $FCorrection(p)$ will be never more satisfied. The only two ways for a processor p to change Pif_p to C are: p satisfies $FAbRoot(p)$, therefore it satisfies $AbRoot(p)$, or p satisfies $Leaf(p)$.*

Lemma 4. *After the first round the root of an abnormal tree can leave it only if the tree is dead.*

Proof. Let ar be the root of an abnormal tree. From Lemma 1, for all p in $Tree(ar)$, $E - Trace(ParentPath(p)) \in B^*C^* \cup B^*F^*$. By Error Algorithm, ar cannot leave the tree until $E_{ar} = F$. So, when ar executes *E-action* and *EC-action*, $E - Trace(ParentPath(p)) \in F^+$ for all p in $Tree(ar)$. By Definition 9 , $Tree(ar)$ is dead.

Lemma 5. *Every processor p such that $PIF_p = C$ in γ satisfies $\neg PFAAR(p)$.*

Proof. If p never moves then the lemma holds. Assume now that p executes *B-action*. Let q be the parent of p, q is in an abnormal tree. $E_p = C$ after p moves. So by Lemma 4, the root of the abnormal tree will leave the tree only when the tree is dead. When p becomes the root of a part of the initial tree, $Tree(p)$ is still dead and the lemma holds.

Lemma 6. *After the first round, every processor p satisfies $\neg PFAAR(p)$.*

Proof. We check the three possible cases after the first round.

(1) If p verifies $Pif_p = C$, by Lemma 5, p satisfies $\neg PFAAR(p)$.

(2) If p is in the normal tree then it must execute *C-action* before to be able to hook on to an abnormal tree. After p executes *C-action*, $Pif_p = C$. Lemma 5 implies that p satisfies $\neg PFAAR(p)$.

(3) If p is in an abnormal tree then Lemma 4 implies that p leaves the tree only if $Tree(p)$ is dead. So, when p leaves the tree, it does not verify $FAAR(p)$, since after this move, p verifies $Pif_p = C$, by the same reasoning as previously, p satisfies $\neg PFAAR(p)$.

Lemma 7. *After the first round all abnormal trees become dead in at most $N-1$ rounds.*

Proof. Corollary 1 implies that after the first round, the root p of an abnormal tree cannot leave it before $E_p = F$. The worst case is obtained when any processor in the tree has the C value in Variable E. So, it is necessary to propagate the B value from the root to the leaves. Then, $h + 1$ rounds are necessary to propagate this value where h is the maximum height of the tree. All the processors different from the root can be in the abnormal tree, this implies that the maximum height is $N - 2$, thus the system needs at most $N - 1$ rounds to propagate B in the tree.

Lemma 8. *In at most $3N-2$ rounds, the system does not contain any abnormal tree.*

Proof. From Lemma 6, after the first round, no processor can leave an abnormal tree and hook on to it again. So any abnormal tree can only contain any processor at most once, and then, disappear once the successive abnormal roots leave it (see Lemma 1). As in proof of Lemma 7, it is necessary to have $N - 1$ rounds to propagate the value F up, and $N-1$ rounds to apply $C-action$ and $EC-action$. After that, all the dead trees have disappear and the system does not contain any abnormal tree in at most $3N - 2$ rounds.

4.3 Proof of Snap-Stabilisation

Lemma 9. *From any initial configuration containing no abnormal tree, the root executes the B-action in at most $5N$ rounds.*

Proof. From these configurations, the worst case is the following: The root has $Pif_r = B$ and all the other processors have their Pif variable equal to C. In this case, the system has to perform a quite complete PIF cycle (a complete cycle except the first step: *B-action* of r). According to the algorithm, *B-action* is propagated to all processors in at most $N-1$ rounds. One extra round is necessary for the leaf processors of the broadcast to set their Que variable to R. The time used by the *QW-action* is bounded by the maximum length of the tree, $N-1$ rounds. By a similar reasoning taking in account that r also executes the respective actions, it is obvious that *QA-action*, *F-action*, and *C-action* need at most N rounds. Furthermore the total time is $5N-1$ rounds, and the root can execute *B-action* during the next round.

By Lemmas 8 and 9, we can claim the following result.

Theorem 1. *From any initial configuration, the root can execute B-action in at most $8N-2$ rounds.*

Lemma 10. *Let p be a processor in an abnormal tree such that $Que_p \in \{Q, R\}$. While p does not leave the tree, $Que_p \neq A$.*

Proof. If p also satisfies $AbRoot(p)$, then p cannot execute *QA-action* before it leaves the tree. Otherwise, if there exists an A value on ParentPath(p), the Q and R values are a barrier for the A. So while p does not leave the tree, it will never receive any A value.

Theorem 2. *From any configuration such that the root executes B-action, the execution satisfies PIF specification.*

Proof. 1. Assume that there exist some processors which never receive the message m sent by r. Then there exists a processor p which never receives m but one of its neighbor (q) does. When q receives m (in configuration γ), it executes the *B-action*. Then, if $Pif_p \neq C$ and $Que_p \in \{W, A\}$, (Pif_q, Que_p) will stay equal to (B, Q) while Que_p equals R, so p eventually executes the *QR-action*. We can remark in this case, that p is in an abnormal tree, since it never receives m. From Lemma 10, $Que_p \neq A$ while p does not leave the tree. While p does not leave the tree, q cannot execute the *F-action*. (q does not satisfy $AnswerOK(q)$ because p). From Lemma 8, p eventually leaves the tree and we reach a configuration where $Pif_p = C$.
While $Pif_p = C$, q cannot execute the *F-action*. From Lemma 8, the system does not contain any abnormal tree in a finite time, so $\forall p'$ neighbor of p, $Par_{p'} \neq p$. So p satisfies $Broadcast(p)$ forever and eventually receives m. (it executes the *B-action*.) We obtain a contradiction.

2. Assume that there exists a processor p, which receives the message m at least twice. p executed the B-, F-, and C-*actions* for m before it satisfies again $Broadcast(p)$ for m. Let P_1 be $ParentPath(p)$ (respectively, P_2) corresponding to the first (respectively, second) reception of m by p. It is clear that Pif_r always equals B while some processor is still in B in the network. So, at least a processor of P_1 has still its Pif variable equal to F. Then $P_2 \neq P_1$. So there exists a path (P_1) from r to p such that $Pif - Trace(P_1) \in B^+F^+C^+$ and there exists a path (P_2) such that the $Pif - Trace(P_2) \in B^+C^+$. In this case, the contradiction is that p was not able to execute B-*action* for the first reception of m, since at least its neighbor in P_2 had its Pif variable equal to C.

We proved that every processor receives m exactly once. (Property *[PIF1]* of Specification 1.) We now show Property *[PIF2]*. Assume that a processor p is such that $Pif_p = B$. Then, it is clear that every processor in ParentPath(p) have their Pif variable equal to B. So, when r executes the F-*action*, the other processors executed F-*action* before and their Pif variable is in $\{C, F\}$.

The execution satisfies PIF specification.

From Remark 1 and Theorems 1 and 2, the following theorem is obvious.

Theorem 3. *The composition of $\mathcal{PIF}$, Question, and Error Algorithms is snap-stabilizing for Specification 1.*

4.4 Complexity Analysis

From Theorem 1 and Lemma 9, we can deduce the following result.

Lemma 11. *From any initial configuration, a complete PIF cycle is executed in at most $13N - 2$ rounds.*

The performances described by Theorem 1 and Lemmas 9 and 11 are the same as those of the previous snap-stabilizing PIF algorithm ([10]) up to a small constant.

5 Conclusion

We presented a new snap-stabilizing PIF algorithm on an arbitrary network. The algorithm does not use a pre-constructed spanning tree. The snap-stabilizing property guarantees that when a processor p initiates the broadcast wave, the broadcast message will reach every processor in the network. Moreover, all the feedback messages correspond to the broadcast message and will be received by p. The snap-stabilizing PIF algorithm presented in this paper improved the solution presented in [10] because it does not need the knowledge of the exact size of the network. So our protocol can be used on dynamic networks. This protocol is a bold step in the comparison of power of self-stabilization and snap-stabilization. It has been proved in [11] that, in the local shared memory model of communication, any static protocol that can be self-stabilized by the extensions of [14] can also be snap-stabilized by extensions using the snap-stabilizing PIF of [10]. With this new PIF algorithm, we conjecture that this result can be extended for dynamic protocols.

References

[1] Y Afek, S Kutten, and M Yung. Memory-efficient self-stabilization on general networks. In *WDAG90 Distributed Algorithms 4th International Workshop Proceedings, Springer-Verlag LNCS:486*, pages 15–28, 1990. 200

[2] L. O. Alima, J. Beauquier, A. K. Datta, and S. Tixeuil. Self-stabilization with global rooted synchronizers. In *ICDCS98 Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 102–109, 1998. 200

[3] A Arora and MG Gouda. Distributed reset. *IEEE Transactions on Computers*, 43:1026–1038, 1994. 200

[4] 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. 200

[5] B Awerbuch, B Patt-Shamir, and G Varghese. Self-stabilization by local checking and correction. In *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 268–277, 1991. 200

[6] B Awerbuch and G Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 258–267, 1991. 200

[7] A Bui, AK Datta, F Petit, and V Villain. State-optimal snap-stabilizing PIF in tree networks. In *Proceedings of the Forth Workshop on Self-Stabilizing Systems*, pages 78–85. IEEE Computer Society Press, 1999. 199, 200

[8] EJH Chang. Echo algorithms: depth parallel operations on general graphs. *IEEE Transactions on Software Engineering*, SE-8:391–401, 1982. 199

[9] A Cournier, AK Datta, F Petit, and V Villain. Self-stabilizing PIF algorithm in arbitrary rooted networks. In *21st International Conference on Distributed Computing Systems (ICDCS-21)*, pages 91–98. IEEE Computer Society Press, 2001. 200

[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. 199, 200, 212, 213

[11] A Cournier, AK Datta, F Petit, and V Villain. Enabling snap-stabilization. In *23rd International Conference on Distributed Computing Systems (ICDCS-23)*. To appear, 2003. 212

[12] EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974. 199

[13] 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. 200, 201

[14] S Katz and KJ Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7:17–26, 1993. 200, 213

[15] HSM Kruijer. Self-stabilization (in spite of distributed control) in tree-structured systems. *Information Processing Letters*, 8:91–95, 1979. 200
[16] N Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996. 199
[17] M Raynal and JM Helary. *Synchronization and Control of Distributed Systems and Programs*. John Wiley and Sons, Chichester, UK, 1990. 199
[18] A Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, IT-29:23–35, 1983. 199
[19] G Tel. *Introduction to distributed algorithms*. Cambridge University Press, 1994. 199
[20] G Varghese. Self-stabilization by local checking and correction (Ph.D. thesis). Technical Report MIT/LCS/TR-583, MIT, 1993. 200