Snap-Stabilizing PIF and Useless Computations*

Alain Cournier Stéphane Devismes Vincent Villain
LaRIA, CNRS FRE 2733
Université de Picardie Jules Verne, Amiens (France)

Abstract

A snap-stabilizing protocol, starting from any configuration, always behaves according to its specification. In other words, a snap-stabilizing protocol is a self-stabilizing protocol which stabilizes in 0 time unit. Here, we propose the first snap-stabilizing propagation of information with feedback for arbitrary networks working with an unfair daemon. An interesting aspect of our solution is that, starting from any configuration, the number of reception (resp. acknowledgement) of corrupted messages (i.e., messages not initiated by the root) by a processor is bounded.

1. Introduction

The concept of *Propagation of Information with Feedback* (PIF) has been introduced by Chang [7] and Segall [13]. The PIF scheme can be described as follows: a node, called *root* or *initiator*, initiates a wave by broadcasting a message *m* into the network (*broadcast phase*). Each non-root processor acknowledges to the root the receipt of *m* (*feedback phase*). The wave terminates when the root has received an acknowledgment from all other processors. In distributed systems, any processor may need to initiate a PIF wave. Thus, any processor can be the initiator of a PIF wave and several PIF protocols may run simultaneously. To cope with the current executions, every processor maintains the identity of the initiators. PIF protocols have been extensively used in distributed systems for solving a wide class of problems, e.g., spanning tree construction, distributed infimum functions computations, snapshot, termination detection, reset, or synchronization. So, designing efficient fault-tolerant PIF protocols is highly desirable. Many fault-tolerant schemes have been proposed and implemented, but the most general technique to design a system tolerating arbitrary transient faults is *self-stabilization* [10]. A self-stabilizing system, regardless of the initial states of the processors and messages initially in the links, is guaranteed to converge into the intended behavior in finite time. Such a property is very desirable because after any unexpected perturbation modifying the messages and/or the memory state, the system eventually recovers without any outside intervention. A particular class of self-stabilizing protocols is *snap-stabilizing* protocols [6]. A *snap-stabilizing* protocol guarantees that, starting from any configuration, it always behaves according to its specification. In other words, a snap-stabilizing protocol is a self-stabilizing protocol which stabilizes in 0 time unit. Of course, a snap-stabilizing protocol is optimal in terms of stabilization time. The notion of 0 stabilization time is a surprising result in the stabilization area. Clearly, the snap-stabilization does not guarantee that all components of the network never work in a fuzzy manner. However, the snap-stabilizing property ensures that if an execution of a protocol is initiated by some processors, then the protocol behaves as expected. Furthermore, after the initialization, the protocol does not introduce any additional fuzzy behavior. Consider, for instance, a snap-stabilizing PIF protocol. Starting from any configuration, the protocol ensures that when a processor p has a message *m* to broadcast, then: p eventually starts the broadcast of *m*, any other processor will receive *m*, and send an acknowledgment which will reach p.

Several PIF protocols for arbitrary networks have been proposed in the self-stabilizing area, e.g., [14, 8]. The self-stabilizing PIF protocols have also been used in the area of self-stabilizing synchronizers [4, 2]. Reset protocols are also PIF-based protocols. Several reset protocols exist in the self-stabilizing literature, e.g., [1, 3]. Self-stabilizing snapshot protocols [12, 14] are also based on the PIF scheme. The first snap-stabilizing PIF protocol for arbitrary networks has been presented in [9]. The drawback of this solution is that the protocol needs to know 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. This drawback is solved in [5].

The complexity analysis of the protocols in [9, 5] reveals that they are efficient in terms of rounds for the stabilization time ($O(1)$), delay ($O(N)$ where N is the number of processors), and execution time ($O(N)$), respectively. How-

*See www.laria.u-picardie.fr/∼devismes/LaRIA-2006-04.pdf for the full version of this paper.

ever, the correctness of these protocols is proven assuming a (weakly) fair daemon only. Roughly speaking, a daemon is considered as an adversary which tries to prevent the protocol to behave as expected, and fairness means that the daemon cannot prevent forever a processor to execute an enabled action. A more (the most) general daemon is the unfair daemon: an unfair daemon can prevent forever an action to be executed except if it is the only enabled action. A well-known property of protocols working under an unfair daemon is that each round of their executions contains a finite number of steps. So, as the protocols in [9, 5] are not proven assuming an unfair daemon, their step complexities cannot a priori be bounded. Therefore, we propose in this paper a novel snap-stabilizing PIF protocol proven assuming an unfair daemon as well as its step complexity analysis. Our protocol is based on the principles presented in [5]. This new solution keeps the advantages of the earlier solutions, i.e., efficient round complexities, low memory requirement, and resilience to dynamic topological changes. However, the snap-stabilizing property of our PIF protocol does not ensure that the network never works in a fuzzy manner. In particular, if we focus on a non-initiator processor p , the snap-stabilization does not ensure that p never receives corrupted messages (i.e., messages not sent by the initiator). Nevertheless, we will see that, using our protocol, the number of corrupted messages that p may receive is bounded by the size of the network (Theorem 8). Also, we will see that p does not acknowledge any corrupted message: the number of these corrupted acknowledgments is bounded by a constant independent of the network (Theorem 9). In many PIF-based applications, the acknowledgments are crucial. For instance, in a distributed infimum functions computation, the computation is distributed during the broadcast and the result is computed and stored into the acknowledgments. So, the cost of local computations generated by acknowledgments of corrupted messages can be very significant. Thus, bounding the number of corrupted acknowledgments by a small constant enhances the quality of the solution.

The rest of the paper is organized as follows: in Section 2, we describe the model we use. In the same section, we define of the snap-stabilization and we give a formal statement of the PIF protocol. In Section 3, we present our PIF protocol. We give a sketch of the proof of snap-stabilization for our protocol and some complexity results in Section 4. Finally, we conclude in Section 5.

2. Computational Model

We consider a network as an undirected connected graph $G = (V,E)$ where V is a set of *processors* and E is the set of *bidirectional asynchronous communication links*. We state that N is the size of G ($|V| = N$) and Δ its degree (i.e., the maximal value among the local degrees of the processors). We assume that the network is *rooted*, i.e., among the processors, we distinguish a particular one, *r*, which is called *root*. We say that p and q are neighbors if and only if (p,q) $\in E$. Each processor p can distinguish all its links. For sake of simplicity, we refer to a link (p,q) of a processor p by the *label* q. We assume that the labels of p, stored in the set $Neig_p$, are locally ordered by $\prec_p$. $Neig_p$ is considered as a constant input from the system. Our protocols are *semi-uniform*, i.e., each processor executes the same program except *r*. We consider a local shared memory model of computation derived from the Dijkstra's state model [10]. Such a model is an abstraction of the message-passing model in a sense that the notion of messages is replaced by the fact that any processor can directly read the state of its neighbors. In our model, the program of every processor consists in a set of *shared variables* (henceforth, referred to as variables) and an *ordered finite set of actions* inducing a *priority*. This priority follows the order of appearance of the actions into the text of the protocol. A processor can write to its own variable only, and read its own variables and that of its neighbors. Each action is constitued as follows: $< label > :: < guard > \rightarrow < statement >$. The guard of an action in the program of p is a boolean expression involving 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 is satisfied. 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. We will refer to the state of a processor and the system as a (*local*) *state* and (*global*) *configuration*, respectively. 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 if and only if the guard of A is satisfied by p in γ. p is said to be *enabled* in γ if and only if at least one action is enabled at p in γ. When several actions are enabled simultaneously at a processor p: only the priority enabled action can be activated. Let a distributed protocol $\mathcal{P}$ be a collection of binary transition relations denoted by $\mapsto$, on $\mathcal{C}$. A *computation* of $\mathcal{P}$ is a *maximal* sequence of configurations $e = \gamma_0,\gamma_1,...,\gamma_i,\gamma_{i+1},...$ 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. *Maximality* means that the sequence is either finite (and no action of $\mathcal{P}$ is enabled in the terminal configuration) or infinite. All computations considered here are assumed to be maximal. $\mathcal{E}$ is the set of all computations of $\mathcal{P}$. As we already said, each execution is decomposed into steps. Each step is shared into three sequential phases atomically executed: (i) every processor evaluates its guards, (ii) a *daemon* (also called *scheduler*) chooses some enabled processors, (iii) each chosen processor executes its priority enabled action. When the three phases are done, the next step begins. A *daemon* can be defined in terms of *fairness* and *distributivity*. We use the

notion of *weakly fairness*: if a daemon is *weakly fair*, then every continuously enabled processor is eventually chosen by the daemon to execute an action. We also use the notion of *unfairness*: the *unfair* daemon can forever prevent a processor to execute an action except if it is the only enabled processor. Concerning the *distributivity*, we assume that the daemon is *distributed* meaning that, at each step, if one or more processors are enabled, then the daemon chooses at least one of these processors to execute an action. We consider that any processor p executed a *disabling 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 protocol action in $\gamma_i \mapsto \gamma_{i+1}$. The disabling action represents the following situation: at least one neighbor of p changes its state in $\gamma_i \mapsto \gamma_{i+1}$, and this change effectively made the guard of all actions of p false in γ_{i+1}. To compute the time complexity, we use the definition of *round* [11]. This definition captures the execution rate of the slowest processor in any computation. Given a computation $e \in \mathcal{E}$, the *first round* of e (let us call it e') is the minimal prefix of e containing the execution of one action (a protocol action or a disabling action) of every enabled processor 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.

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

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 $\mathcal{SP}_{\mathcal{T}}$ if and only if $\forall e \in \mathcal{E} :: e \vdash \mathcal{SP}_{\mathcal{T}}$.*

The problem to be solved. Any processor can be an initiator in a PIF and several PIF protocols may run concurrently. Here, we consider the problem in a general setting of the PIF scheme where the PIF is initiated by *r* only.

Specification 1 (PIF Wave) *A finite computation $e = \gamma_0, \ldots, \gamma_i, \gamma_{i+1}, \ldots, \gamma_t \in \mathcal{E}$ is called a* PIF Wave *if and only if the following condition is true:*
If r *broadcasts a message* m *in the 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 *To prove that a PIF protocol is snap-stabilizing we must show that any execution of the protocol satisfies these two conditions: (i) if* r *has a message* m *to broadcast, it will do in a finite time, and (ii) from any configuration where* r *broadcasts* m*, the system satisfies Specification 1.*

3. Algorithm

We now describe our snap-stabilizing PIF protocol (Algorithms 1 and 2) referred to as Algorithm $\mathcal{PIF}$. Algorithm $\mathcal{PIF}$ is divided into three parts:

- The *PIF Part*. This is the main part of the protocol. This part contains the actions corresponding to each of the three phases of the PIF: the broadcast phase, the feedback phase following the broadcast phase, and the cleaning phase which cleans the trace of the feedback phase so that *r* is ready to broadcast a new message.
- The *Question Part*. This part ensures that each processor eventually receives the message from *r* during a broadcast phase. Especially when the system contains erroneous behaviors (the system can start from any configuration). Actually, the question part controls that, after receiving a message from *r*, a processor does not execute the feedback phase before all its neighbors received the message.
- The *Correction Part*. This part contains the actions dealing with the error correction, i.e., the actions that clean the erroneous behaviors.

We now more precisely describe these three parts.

PIF Part. Let $\gamma \in \mathcal{C}$ where $\forall p \in V$, $S_p = C$, referred to as the *normal starting configuration*. In γ, $B\text{-}action$ at *r* is the only enabled action of the system. So, *r* executes $B\text{-}action$ in the first step: *r* broadcasts a message *m*, switches to the broadcast phase by $S_r := B$, and initiates a *question* by $Que_r := Q$ (we will see later what the goal of this question is). When a processor p waiting for a message (i.e., $S_p = C$) finds one of its neighbors q in the broadcast phase ($S_q = B$), p receives the message from q ($B\text{-}action$): it also switches to the broadcast phase ($S_p := B$), initiates a question ($Que_p := Q$), points out to q using P_p, and sets its level L_p to $L_q + 1$. Typically, L_p contains the length of the path followed by the broadcast message from *r* to p. (Since *r* never receives any broadcast message from any neighbor, P_r and L_r are constants.) p is now in the broadcast phase ($S_p = B$) and is supposed to broadcast the message to its neighbors except P_p. So, step by step, a spanning tree rooted at *r* (w.r.t. the P variables), noted $Tree(r)$, is dynamically built during the broadcast phase. Eventually, some processor p in $Tree(r)$ cannot broadcast the message because all its neighbors have already received the message from some other neighbors ($\forall q \in Neig_p$, $S_q \neq C \wedge P_q \neq p$). Then, p (called a *leaf* of $Tree(r)$) waits an authorization from the root to execute the feedback phase. This authorization corresponds to the reception by p and its neighbors of an answer to the question previously asked by p ($AnswerOk(p)$). After receiving this authorization, p can switch to the feedback phase by $F\text{-}action$ ($S_p := F$). The

Algorithm 1 $\mathcal{PIF}$ for $p = r$

Input: $Neig_p$: set of (locally) ordered neighbors of p;

Constants: $P_p = \perp$; $L_p = 0$;

Variables: $S_p \in \{B,F,P,C\}$; $Que_p \in \{Q,R,A\}$;

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

Predicates:

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

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

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

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

$Broadcast(p) \equiv (S_p = C) \wedge Leaf(p)$

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

$PreClean(p) \equiv (S_p = F) \wedge [\forall q \in Neig_p :: (P_q = p) \Rightarrow (S_q \in \{F,C\})]$

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

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

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

Actions:

PIF Part:

$B\text{-}action :: Broadcast(p) \rightarrow S_p := B; Que_p := Q;$ /* Initialization Action */

$F\text{-}action :: Feedback(p) \rightarrow S_p := F;$

$P\text{-}action :: PreClean(p) \rightarrow S_p := P;$

$C\text{-}action :: Cleaning(p) \rightarrow S_p := C;$

Question Part:

$QR\text{-}action :: Require(p) \rightarrow Que_p := R;$

$QA\text{-}action :: Answer(p) \rightarrow Que_p := A;$

feedback phase is then propagated up into $Tree(r)$ as follows: a non-leaf processor q switches to the feedback phase when (i) it is authorized by r ($AnswerOk(q)$), (ii) all its neighbors satisfy $S \neq C$ ($CFree(q)$), and (iii) all its children in $Tree(r)$ satisfy $S = F$ ($BLeaf(q)$). By this mechanism, all processor eventually participates to both broadcast and feedback phase. Only r detects the end of the feedback phase: when setting S_r to F. It then remains to execute the last phase to the PIF wave: the cleaning phase. The aim of this phase is to clean the trace of the PIF wave to bring the system in the normal starting configuration again. This phase works as follows. Since r detects the end of the feedback phase ($S_r = F$), r sets S_r to P. This value is then propagated in $Tree(r)$ toward its leaves ($PC\text{-}action$) to inform all processors of the termination. Finally, after receiving the P value, each successive leaf cleans itself by setting its S variable to C ($C\text{-}action$). Therefore, the system will reach the normal starting configuration again.

Question Part. We saw that when a processor p switches to the broadcast phase ($S_p := B$), it also initiates a *question* ($Que_p := Q$). Then, since $S_p = B$, p waits an autorization before executing its feedback phase ($S_p := F$). This autorization corresponds to the reception by p and its neighbors of an answer to the question previously asked by p. The questions are used for providing the following problem. When the system starts from a configuration different of γ (the system can start from any configuration), some neighbors of p, q, may satisfy $S_q \in \{B,F\}$ while they are not in $Tree(r)$. Actually, these processors are in trees rooted by some other processors than r: *abnormal trees*. We will see later (next paragraph) that these abnormal trees are eventually erased from the system using the *Correction Part*. But, while such a processor q is in an abnormal tree, p must not switches to the feedback phase. Otherwise, q does not receive the broadcast message from p and, as a consequence, q may never receive the message sent by r. That is why we use the *questions*. The goal of the question (and its respective answers) is to ensure that p switches to the feedback phase only when all its neighbors are in $Tree(r)$. Of course, the neighbors of p are in $Tree(r)$ since they received the message from r. The question mechanism is managed into the Que variables: $Que_p \in \{Q,R,A\}$ for $p = r$ and $Que_p \in \{Q,R,W,A\}$ for $p \neq r$. The Q and R variables are used for resetting the part of the network which is concerned by a *question*. The W value corresponds to the request of a processor: "Do you authorize me to feedback?". The A value corresponds to the answer sending by r (n.b., r is the only processor able to generate a A value). We now explain how this phase works. We already saw that a *question* is initiated at $p \in V$ by $Que_p := Q$ each time p switches to a broadcast phase. This action forces all its neighbor q satisfying $S_q \neq C$ to execute $Que_q := R$ ($QR\text{-}action$). When every q has reset, p also executes $QR\text{-}action$. The R values are then propagated up in the trees of p and each q (and only these trees) following the P variables. By this mechanism, all A values possibly in the path from p (resp. q) to its root (w.r.t. the P variable) are erased (in particular, the A value present since the initial configuration). So, from now on, when a A value reaches a requesting processor or one of its neighbor, this value cannot come from any one but r and the processor obviously is in $Tree(r)$. Then,

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

Input: $Neig_p$: set of (locally) ordered neighbors of p;

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

Macros:

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

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

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

Predicates:

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

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

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

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

$GoodS(p) \equiv (S_p = C) \vee [(S_{P_p} \neq S_p) \Rightarrow ((S_{P_p} = EB) \vee (S_p = F \wedge S_{P_p} \in \{B,P\}))]$

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

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

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

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

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

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

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

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

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

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

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

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

Actions:

Correction Part:

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

$EB\text{-}action :: EBroadcast(p) \rightarrow S_p := EB;$

$EF\text{-}action :: EFeedback(p) \rightarrow S_p := EF;$

PIF Part:

$B\text{-}action :: Broadcast(p) \rightarrow S_p := B; P_p := \min_{\prec_p}(Potential_p); L_p := L_{P_p} + 1; Que_p := Q;$

$F\text{-}action :: Feedback(p) \rightarrow S_p := F;$

$P\text{-}action :: PreClean(p) \rightarrow S_p := P;$

$C\text{-}action :: Cleaning(p) \rightarrow S_p := C;$

Question Part:

$QR\text{-}action :: Require(p) \rightarrow Que_p := R;$

$QW\text{-}action :: Wait(p) \rightarrow Que_p := W;$

$QA\text{-}action :: Answer(p) \rightarrow Que_p := A;$

as we have seen before, eventually, some processors p in $Tree(r)$, called leaves, detect that the broadcast phase cannot progress anymore because all their neighbors q have received a broadcast message from some other processors (i.e., $S_q \neq C$ and $P_q \neq p$, $\forall q$). In this case, p executes $Que_p := W$ ($QW\text{-}action$) meaning that it is now waiting for an answer from *r*. The W value is then propagated up into the tree of p (and only this tree) as follows: a non-leaf processor q can execute $QW\text{-}action$ if all its children have set their Que variable to W and no neighbor has still $S = C$. When the W values reaches all the children of *r*, *r* executes $QA\text{-}action$: *r* broadcasts an answer A into its tree and so on. So, $\forall p \in V$, after initiating a question (i.e., $Que_p := Q$), if p and its neigbors are in $Tree(r)$, then they eventually receive a A value. In this case, p is sure that itself and all its neighbor are in $Tree(r)$ and is authorized to execute $F\text{-}action$ ($AnswerOk(p)$). Otherwise, the processors in an abnormal tree receive no A until they leave their trees (using the *Correction Part*) and hook on to the normal tree $Tree(r)$. In particular, if p is in $Tree(r)$, its $F\text{-}action$ will be enabled only after its neighbors hook on to $Tree(r)$.

Correction Part. This part is used for erasing the erroneous behaviors. Of course, the error correction only concerns processors p such that $S_p \neq C$ and $p \notin Tree(r)$, i.e., the *abnormal processors*. The abnormal processors are arranged into abnormal trees rooted at a processor satisfying $AbRoot$, i.e., an *abnormal root*. We define the abnormal trees as follows: let p such that $AbRoot(p)$, $\forall q \in V$, $q \in Tree(p)$ (the abnormal tree rooted at p) if and only if there exists a sequence of processors ($p_0 = p$), ..., p_i, ..., ($p_k = q$) such that, $\forall i \in [1...k]$, $p_i \in Child_{p_{i-1}}$ (among the neighbors designating p_{i-1} with P only those satisfying $S \neq C \wedge \neg AbRoot$ are considered as p_{i-1} children). So, the error correction consists in the removal of all these abnormal trees. To remove an abnormal tree $Tree(p)$, we cannot simply set S_p to C. Since some processor can be in $Tree(p)$. If we set S_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, this scheme can progress infinitely often, and the system may contain forever an abnormal tree which can prevent the progression of the normal tree $Tree(r)$. We solve this problem by paralyzing the process of any abnormal tree before

remove it. To that goal, we use two additional states in the S variables: EB and EF (for $p \neq r$ only). If p is an abnormal root, it sets its variable S_p to EB and broadcasts this value into its tree and only its tree (EB-$action$). When p receives an acknowledgment (EF-$action$) of all its children (value EF of variable S), p knows that all the processors q of its tree satisfy $E_q = EF$ and no processor can now receive the broadcast phase from any q (indeed, $S_q \neq B, \forall q$). Then p can leave its tree (EC-$action$) and it will try to receive the broadcast from one of the processors q only when q participates in another broadcast. By this process, all abnormal trees eventually disappear and $Tree(r)$ will be able to grow until it reaches all the processors of the network.

4. Correctness and Complexity Analysis

To prove that $\mathcal{PIF}$ is snap-stabilizing for Specification 1 under an unfair daemon we used the following scheme of proof: we first prove that $\mathcal{PIF}$ is snap-stabilizing for a weakly fair daemon; we then prove that each PIF wave is executed in a finite number of steps.

Some Definitions. $\forall p \in V$ such that $S_p \neq C$, $PPath(p)$ is the unique path $p_0, p_1, p_2, \ldots,(p_k = p)$ satisfying these two conditions: (i) $\forall i$, $1 \leq i \leq k$, $(S_{p_i} \neq C) \wedge (P_{p_i} = p_{i-1}) \wedge \neg AbRoot(p_i)$, (ii) $(p_0 = r) \vee AbRoot(p_0)$. $\forall p \in V$ such that $(p = r) \vee AbRoot(p)$, we define a set $Tree(p)$ of processors as follows: $\forall q \in V$, $q \in Tree(p)$ if and only if $S_q \neq C$ and p is the first extremity of $PPath(q)$. A tree containing only processors p such that $(p = r) \vee \neg AbRoot(p)$ is called a *normal tree*. Obviously, the system always contains one normal tree: the tree rooted at *r*. Any tree rooted at another processor than *r* is called an *abnormal tree*. A tree T satisfies $Alive(T)$ (or is called $Alive$) if and only if $\exists p \in T$ such that $S_p = B$. A tree T satisfies $Dead(T)$ (or is called $Dead$) if and only if $\neg Alive(T)$.

Proof assuming a Weakly Fair Daemon.

Lemma 1 *The system contains no abnormal tree in at most* $3N - 3$ *rounds.*

Proof Outline. The *Correction Part* acts upon the abnormal trees as follows: First, the EB values are propagated down into the trees until their leaves by EB-$action$ in at most $h + 1$ rounds where h is the maximal height of an abnormal tree. Then, also $h+1$ rounds are necessary to propagate up the EF values (EF-$action$) to the abnormal roots. From this point out, all abnormal trees are dead and still $h + 1$ rounds are necessary so that the abnormal trees disappear by the successive removing of abnormal roots (EC-$action$). Now, by definition, all non-root processors can be into an abnormal tree. This implies that the maximal height of such trees is $N - 2$ and the lemma holds. □

Lemma 2 *From any configuration containing no abnormal tree,* r *executes* B-$action$ *in at most* $6N$ *rounds.*

Proof Outline. Clearly, from such configurations, the worst case is the following: *r* satisfies $S_r = B$ and all the other processors have their S variable equal to C. According to the algorithm, B-$action$ is propagated to all processors in at most $N - 1$ rounds. (Note that after executing B-$action$, a processor is enabled to executed QR-$action$, so B-$actions$ and QR-$actions$ work in parallel.) After all processors executed their B-$action$, one extra round is necessary for the leaves to set their Que variable to R. Then, the W value is propagated up in the into the Que variables by QW-$action$. The time used by the QW-$actions$ is bounded by $N - 1$ rounds. By a similar reasonning taking in account that *r* also executes the respective actions, it is obvious that the QA-$actions$, F-$actions$, P-$actions$, and C-$actions$ are successively propagated into the tree in at most N rounds each one. Hence, after $6N - 1$ rounds, the system reaches a configuration where $\forall p \in V$, $S_p = C$ and *r* executes B-$action$ in the next round. □

By Lemmas 1 and 2, the following result holds.

Theorem 1 *From any configuration,* r *execute* B-$action$ *in at most* $9N - 3$ *rounds.*

Theorem 2 *From any configuration where* r *executes* B-$action$*, the execution satisfies Specification 1.*

Proof Outline. From such a configuration, we know that the abnormal trees cannot prevent forever the PIF from *r* to progress (Lemma 1). Also, from the explanation provided in Section 3 (*Question Part*), we know that any processor p which has received the message initiated by *r* (B-$action$) cannot switch to the feedback phase before all its neighbors also receive this message. This implies that every processor are eventually into the normal tree and [PIF1] is satisfied. Also, since they are into the normal tree, each p cannot leave it before *r* initiates the cleaning phase by P-$action$. Now, *r* initiates this phase only when $S_r = F$ and *r* sets S_r to F only when all its neighbors are into the normal tree and all its children are into the feedback phase. Inductively, it is easy to verify that the children q of *r* into the normal tree also switch S_q to F only when their neighbors are into the normal tree and their children are into the feedback phase and so on. Hence, we can conclude that *r* sets S_r to F only when all the other processors have acknowledged its message to it ([PIF2]). □

By Remark 1, Theorems 1, and 2, follows:

Theorem 3 *$\mathcal{PIF}$ is snap-stabilizing for Specification 1 under a weakly fair daemon.*

Proof assuming an Unfair Daemon. To prove the snap-stabilization of $\mathcal{PIF}$ for an unfair daemon, it remains to show that any PIF Wave is finite in terms of steps.

Lemma 3 *$\forall p \in V \setminus \{\mathrm{r}\}$, if p hooks on to an abnormal tree, p cannot execute F-$action$ before leaving the tree.*

Proof Outline. A processor p hooks on to an abnormal tree by B-$action$. By B-$action$, it also initiates a question ($Que_p := Q$). Now, from Section 3 (*Question Part*) we know that p never receives an answer ($Que_p := A$) while it is in the tree. So, while it is in the tree, p never satisfies $AnswerOk(p)$ and F-$action$ is disabled at p. □

A processor p in an abnormal tree can execute F-, P-, and C-$actions$ at most once before leaving the tree. Now, Lemma 3 implies that, after leaving the tree, F-$action$ at p will be no more enabled while p is in an abnormal tree. Another consequence is that P-, and C-$action$ will be also no more enabled while p is in an abnormal tree. Hence, during the whole execution, each processor executes F-, P-, and C-$action$ at most once while it is in an abnormal tree, i.e., $O(N)$ F-, P-, and C-$actions$ are executed on the abnormal trees during the whole execution. Lemma 3 also implies that any processor p which hooks on to an abnormal tree will leave it only by executing the actions of the *Correction Part*. Now, the *Correction Part* ensures that p will leave the tree only when it is dead. Thus, in the worst case, each non-root processor ($N-1$) can hook on to each abnormal tree ($N-1$) at most once during the execution, i.e., $O(N^2)$ B-$actions$. Hence, follows:

Lemma 4 *In an execution, $O(N^2)$ actions of the* PIF Part *are executed on the abnormal trees.*

We now focus on the *Question Part*. A question is initiated at p when it hooks on to a tree ($Que_p := Q$). Then, R values are propagated up into the $PPaths$ of p and its neighbors q such that $S_q \neq C$ ($O(\Delta)$ processors). Then, in the worst case, W values are also propagated up into the $PPaths$ of p and its neighbors q such that $S_q \neq C$. Finally, A values are propagated from *r* to the processor among p and its neighbors that effectively belong to the normal tree. Now, as the height of any $PPath$ is in $O(N)$, follows:

Lemma 5 *Each B-$action$ generates $O(\Delta \times N)$ actions of the* Question Part.

Lemma 6 *In an execution, the abnormal trees generate an overcost of $O(\Delta \times N^3)$ actions of the* Question Part.

Proof Outline. A processor propagates a question in trees because of the initial configuration or when it hooks on to a tree. Each time it hooks on to an abnormal tree, it generates $O(\Delta \times N)$ actions of the *Question Part* (Lemma 5). Of course, the number of actions of the *Question Part* generated if it is in abnormal tree since the initial configuration is of the same order ($O(\Delta \times N)$). Now, $O(N)$ processors are in abnormal trees at the initial configuration and $O(N^2)$ processors hook on to abnormal trees in the execution. □

Finally, we count the actions of the *Correction Part*.

Lemma 7 *In an execution, $O(N^2)$ actions of the* Correction Part *are executed on the abnormal trees.*

Proof Outline. In the worst case, each processor in abnormal trees has to execute the three actions of the *Correction Part* to leave its tree. So, any processor leaves an abnormal tree in $O(1)$ actions of the *Correction Part*. Now, $O(N)$ processors are in abnormal trees at the initial configuration and $O(N^2)$ processors hook on to abnormal trees during the whole execution. □

From Lemmas 4, 6, and 7, we can deduce this result:

Lemma 8 *In an execution, the abnormal trees generate an overcost of $O(\Delta \times N^3)$ actions before disappearing.*

We now show that, starting from any configuration, the normal tree, $Tree(r)$, can only generate a finite number of actions before *r* initiates a PIF wave (by B-$action$). First, from Section 3 (*PIF Part*), we can deduce this result:

Lemma 9 *From any configuration, $O(N)$ actions of the* PIF Part *are executed on $Tree(\mathrm{r})$ before* r *executes B-$action$.*

Following the same reasonning as for Lemma 6, we can easily show the next result:

Lemma 10 *From any configuration, $Tree(\mathrm{r})$ generates $O(\Delta \times N^2)$ actions of the* Question Part *before* r *executes B-$action$.*

By Lemma 9, and 10, follows:

Lemma 11 *From any configuration, $Tree(\mathrm{r})$ generates $O(\Delta \times N^2)$ actions before* r *executes B-$action$.*

By Lemmas 8 and 11, follows:

Theorem 4 *From any configuration,* r *executes B-$action$ after $O(\Delta \times N^3)$ steps.*

Corollary 1 *From any configuration, a complete wave of $\mathcal{PIF}$ is executed in $O(\Delta \times N^3)$ steps.*

By Theorems 3 and Corollary 1, follows:

Theorem 5 *$\mathcal{PIF}$ is snap-stabilizing for Specification 1 under the unfair daemon.*

Space Complexity. It is easy to see that $\mathcal{PIF}$ remains valid if we bound the maximal value of L by N. So, we can claim that any L variable can be stored in $O(\log(N))$ and, by taking account of the other variables, follows:

Theorem 6 *The space requirement of $\mathcal{PIF}$ is $O(\log(N) + \log(\Delta))$ bits per processor.*

Time Complexity. The following results complete the complexity analysis. The first one can be deduce from Theorem 1, Lemmas 1 and 2.

Theorem 7 *From any initial configuration, a complete PIF wave is executed in at most $15N - 3$ rounds.*

We have seen that a processor execute B-$action$ to hook on to abnormal trees at most $N - 1$ times. Moreover, it is easy to see that it can hook on to the normal tree by B-$action$ only once before *r* initiates the protocol. Hence, follows:

Theorem 8 *Starting from any configuration, $\forall p \in V$, p can receive $O(N)$ corrupted messages.*

We have seen that, during the whole execution, a processor can execute F-$action$ at most once while it is into an abnormal tree. Moreover, it is easy to see that a processor of the normal tree can execute F-$action$ only once before *r* initiates the protocol. Hence, follows:

Theorem 9 *Starting from any configuration, $\forall p \in V$, p acknowledges at most twice corrupted messages.*

5 Conclusion

We proposed the first snap-stabilizing PIF for arbitrary rooted networks proven assuming an unfair daemon. The protocol does not need any pre-computed spanning tree as well as the knowledge of the size of the network. The memory requirement of our solution is equivalent to those of [9, 5] ($O(\log(N) + \log(\Delta))$ bits per processor). In contrast with the previous snap-stabilizing solutions ([9, 5]), we can now evaluate the step complexities of our protocols. From any configuration, the protocol starts in $O(\Delta \times N^3)$ steps (resp. at most $9N - 3$ rounds). Also, using our protocol, a complete PIF wave is executed in $O(\Delta \times N^3)$ steps (resp. at most $15N - 3$ rounds). The round complexities of our solution are also equivalent to those of [9, 5]. Finally, another desirable property of our solution is that, starting from any configuration, the number of corrupted messages (i.e., messages not sent by *r*) that a processor may acknowledge is bounded by two.

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.

[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.

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

[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.

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

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

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

[8] A. Cournier, A. 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.

[9] A. Cournier, A. 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.

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

[11] 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.

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

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

[14] G. Varghese. Self-stabilization by local checking and correction (Ph.D. thesis). Technical Report MIT/LCS/TR-583, MIT, 1993.