Snap-Stabilizing PIF Algorithm in Arbitrary Networks

Alain Cournier[1] Ajoy K. Datta[2] Franck Petit[1] Vincent Villain[1]
[1] LaRIA, Université de Picardie Jules Verne, Amiens France
{cournier,petit,villain}@laria.u-picardie.fr
[2] Department of Computer Science, University of Nevada Las Vegas
datta@faculty.egr.unlv.edu

Abstract

We present the first snap-stabilizing Propagation of Information with Feedback (PIF) protocol in arbitrary networks. A snap-stabilizing protocol*, starting from any arbitrary initial system configuration, always behaves according to its specification. Our protocol is distributed, deterministic, and does not use a pre-constructed spanning tree.*
Keywords: *Fault-tolerance, propagation of information with feedback, reset protocols, self-stabilization, snap-stabilization, wave algorithms.*

1 Introduction

Chang [10] and Segall [21] 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. In arbitrary distributed systems, any processor may need to initiate a global computation. Thus, any processor can be an initiator in a PIF protocol, and several PIF protocols may be running simultaneously. To cope with this concurrent execution of the PIF algorithms, every processor maintains the identity of the initiators. 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 [19], [20], and [22] for details). So, designing efficient fault-tolerant wave algorithms is an important task in the distributed computing research. The concept of *self-stabilization* [14] 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 [9]. 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, 8, 9, 16, 18] for tree networks, and [12, 23] 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, 23]). Self-stabilizing snapshot algorithms [17, 23] are also based on PIF scheme. Snap-Stabilizing PIF for oriented and unoriented tree networks are proposed in [7, 9]. The PIF algorithms for trees of [7, 9] are also optimal in terms of space. Except in [12, 23], all self-stabilizing PIF algorithms in the current literature work on trees. These protocols assume an underlying self-stabilizing rooted spanning tree construction algorithm [1, 3, 4, 11, 15]. So, to design a reset or snapshot protocol using these self-stabilizing PIF algorithms, PIF algorithms must be modified such that every processor sends messages to all its outgoing links including the links which are not in the spanning tree.

Contribution. We present a snap-stabilizing PIF algorithm on an arbitrary network. The wave scheme in this paper is a PIF scheme for an arbitrary graph which does not use a pre-constructed spanning tree. The property of snap-stabilization of the proposed algorithm ensures that the protocol *always* works as expected (by its specification). On

Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)
1063-6927/02 $17.00 © 2002 IEEE

the contrary, a self-stabilizing algorithm [12, 23] achieves the convergence to the specified behavior of the system *only* in a finite time. So, using a self-stabilizing algorithm, when a processor p starts a PIF wave to propagate a value, say V, it is not guaranteed that every processor will receive V. In other words, a self-stabilizing algorithm just guarantees that, eventually, a value (not necessary V) propagated with a PIF wave initiated by p will be received by every processor. Removing this particular drawback is the goal of our snap-stabilizing PIF. In the same situation as above, i.e., when p starts a PIF wave to propagate a value V, using the proposed PIF protocol, the property of snap-stabilization ensures that every processor receives V and sends an acknowledgment which will reach p.

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 2. We prove the correctness of the algorithm in Section 4, followed by the complexity analysis (Section 5). Finally, we make some concluding remarks in Section 6.

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. We assume that the labels, stored in the set $Neig_p$, are arranged in some arbitrary order $\succ_p$ $(\forall q_1, q_2 \in Neig_p :: (q_1 \succ_p q_2) \wedge (q_2 \succ_p q_1) \iff (q_1 = q_2))$. 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: $< label >:: < guard > - \rightarrow < statement >$. 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 $e = \gamma_0, \gamma_1, ..., \gamma_i, \gamma_{i+1}, ...$, 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 compute the time complexity measure, we use the definition of *round* [16]. 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., $e = e'e''$. Then *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 $\mathcal{X}$.

Definition 1 (Snap-stabilization) *Let $\mathcal{T}$ be a task, and $\mathcal{SP}_{\mathcal{T}}$ the specification of $\mathcal{T}$. The protocol $\mathcal{P}$ is snap-stabilizing for the specification $\mathcal{SP}_{\mathcal{T}}$ on $\mathcal{E}$ iff 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.

Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)
1063-6927/02 $17.00 © 2002 IEEE

Definition 2 (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 iff Processor* r *broadcasting a message* m *in the computation step* $\gamma_0 \mapsto \gamma_1$ *implies:*

[PIF1] *For each* $p \neq r$*, there exists* $i \in [1, t-1]$ *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 *Every finite computation in which* r *broadcasts no message* m *is also a PIF Cycle.*

Specification 1 (PIF Scheme) *The* PIF scheme *is an infinite sequence of PIF Cycles.*

From Specification 1 and Remark 1, before the root broadcasts any message, the system is in a PIF Cycle. So, every execution e with a prefix α in which r sends no messages, followed by a suffix β such that r sends a message in the first computation step, is a PIF scheme, provided Conditions [PIF1] and [PIF2] are satisfied.

3 Algorithm

3.1 Normal Behavior

The snap-stabilizing PIF algorithm is shown in Algorithms 1 and 2 for the root and other processors, respectively. Let us quickly review the details of the PIF scheme. The PIF scheme is the repetition of the PIF Cycle. The PIF cycle can be informally described as follows: Starting from an initial configuration where no message has yet been broadcast, the root r initiates the *broadcast* phase. The neighbors of r participate in this phase by forwarding the broadcast message, if possible. It is not possible for a processor p to broadcast the message if all its neighbors have received the message from some other neighbor. So, a spanning tree rooted at r is dynamically built during the broadcast phase. Let us call this tree the $B\text{-}tree_r$. The processors which are not able to broadcast the message further are the leaves of $B\text{-}tree_r$. Once the broadcast phase reaches the leaf processors of $B\text{-}tree_r$, they notify to their parent in $B\text{-}tree_r$ of the termination of the broadcast phase by initiating the *feedback* phase. The feedback phase eventually reaches the root r. This completes the current PIF Cycle. Based on the above description, each processor p executes at least two actions. The first one, called the *B-action*, refers to the action executed by p during the *broadcast* phase. The second one is called the *F-action*. *F-actions* refer to the action executed during the *feedback* phase. Every processor maintains a variable Pif_p. This variable can take three different values as described below:

C: p is ready to participate in the next PIF Cycle.

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

Input: $Neig_p$: set of (locally) ordered neighbors
N : number of processors in the network;
$L_{max} : \geq N - 1$

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

Variables:
$Count_p \in [1, N']$, where N' is an upper bound of N;
Fok_p : Boolean;
$Pif_p \in \{B, F, C\}$

Macros:
$Sum\_Set_p = \{q \in Neig_p :: (Pif_q = B) \wedge (Par_q = p) \wedge (L_q = L_p + 1) \wedge \neg Fok_p\}$;
$Sum_p = 1 + \sum_{q \in Sum\_Set_p} Count_q$;

Predicates:
$GoodFok(p) \equiv (Pif_p = B) \Rightarrow (Fok_p = (Sum_p = N))$;
$GoodCount(p) \equiv ((Pif_p = B) \wedge \neg Fok_p) \Rightarrow (Count_p \leq Sum_p)$;
$Normal(p) \equiv GoodFok(p) \wedge GoodCount(p)$;
$Broadcast(p) \equiv (Pif_p = C) \wedge (\forall q \in Neig_p :: Pif_q = C)$;
$Feedback(p) \equiv (Pif_p = B) \wedge Normal(p) \wedge (\forall q \in Neig_p :: Pif_q \neq B) \wedge Fok_p$;
$Cleaning(p) \equiv (Pif_p = F) \wedge (\forall q \in Neig_p :: Pif_q = C)$
$NewCount(p) \equiv (Pif_p = B) \wedge Normal(p) \wedge (Count_p < Sum_p) \wedge \neg Fok_p$;

Actions:
$B\text{-}action :: Broadcast(p) \longrightarrow Pif_p := B; Count_p := 1; Fok_p := (1 = N)$;
$F\text{-}action :: Feedback(p) \longrightarrow Pif_p := F$;
$C\text{-}action :: Cleaning(p) \longrightarrow Pif_p := C$;
$Count\text{-}action :: NewCount(p) \longrightarrow Count_p := Sum_p; Fok_p := (Sum_p = N)$;
$B\text{-}correction :: \neg Normal(p) \longrightarrow Pif_p := C$;

B: p sent the broadcast message if $p = r$. Otherwise ($\neq r$), p has received a message from one of its neighbors (Par_p as described below) and has broadcast this message to its (p's) neighbors, except Par_p.

F: If p is not a leaf processor in $B\text{-}tree_r$, p received a feedback from all its neighbors p sent a message to, meaning that all of them received the message broadcast by p. Each processor $p \neq r$ acknowledges to its parent in $B\text{-}tree_r$ of the receipt of the message.

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 and switches to the broadcast phase by executing $Pif_r = B$ ($B\text{-}action$). 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 $L_q + 1$ ($B\text{-}action$). 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, Par_r and L_r are shown as **constants** 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). Variable $Count_p$ holds the number of nodes which are involved in the subtree dynamically built during the broadcast phase from p, i.e., $B\text{-}tree_p$. When p receives the message broadcast by the root, p sets $Count_p$ to 1. Next, each time a neighbor q of p takes p as the parent, p re-computes $Count_p$ (*Count-action*). But, p may not be able to broadcast the message further. In that case, p is a leaf processor in $B\text{-}tree_r$, and $Count_p$ remains equal to 1 ($B\text{-}tree_p = \{p\}$).

Since every processor p (including r) maintains the variable $Count_p$, $Count_r$ contains the total number of processors currently involved in $B\text{-}tree_r$. So, $Count_r$ is eventually equal to N, the total number of processors in the network. (N is considered as an input at the root.) Once $Count_r = N$, r initiates a wave in $B\text{-}tree_r$. This wave is implemented by using the boolean variable Fok_p maintained by every processor p ($Fok\text{-}action$). The Fok wave allows every leaf processor p in $B\text{-}tree_r$ to initiate the feedback phase. So, when a leaf processor in $B\text{-}tree$ is reached by the Fok wave, p switches to the feedback phase by setting Pif_p to F ($F\text{-}action$). Every processor p propagates the feedback phase towards the root in $B\text{-}tree_r$ by executing the action $F\text{-}action$. 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 back to the normal starting configuration ($\forall p, Pif_p = C$). The *C-action* refers to the action executed by a processor during the cleaning phase. 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 in either the feedback phase ($Pif_q = F$) or 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. Then, the system is in the normal starting configuration again. The root is now ready to start a new PIF cycle. The snap-stabilization of the algorithm is guaranteed by the knowledge of the exact size of the network (N) at the root.

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 $p \neq r$, 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 $GoodPif$).

Algorithm 2 ($\mathcal{PIF}$) For other processors ($p \neq r$).

Input: $Neig_p$: set of (locally) ordered neighbors
$L_{max} : \geq N - 1$

Variables:
Fok_p : Boolean;
$Count_p \in [1, N']$, where N' is an upper bound of N;
$Pif_p \in \{B, F, C\}$
$L_p \in [1, L_{max}]$;
$Par_p \in Neig_p$ if $p \neq r$;

Macros:
$Sum\_Set_p = \{q \in Neig_p :: (Pif_q = B) \wedge (Par_q = p) \wedge (L_q = L_p + 1) \wedge \neg Fok_p\}$;
$Sum_p = 1 + \sum_{q \in Sum\_Set_p} Count_q$;
$Pre\_Potential_p = \{q \in Neig_p :: (Pif_q = B) \wedge (Par_q \neq p) \wedge (L_q < L_{max}) \wedge \neg Fok_q\}$;
$Potential_p = \{q \in Pre\_Potential_p :: \forall u \in Set_p,\ L_u \geq L_q\}$

Predicates:
$GoodFok(p) \equiv ((Pif_p = B) \Rightarrow ((Fok_p \neq Fok_{Par_p}) \Rightarrow \neg Fok_p)) \wedge ((Pif_p = F) \Rightarrow ((Pif_{Par_p} = B) \Rightarrow Fok_{Par_p}))$
$GoodPif(p) \equiv (Pif_p \neq C) \Rightarrow ((Pif_{Par_p} \neq Pif_p) \Rightarrow (Pif_{Par_p} = B))$;
$GoodLevel(p) \equiv (Pif_p \neq C) \Rightarrow (L_p = L_{Par_p} + 1)$;
$GoodCount(p) \equiv ((Pif_p = B) \wedge \neg Fok_p) \Rightarrow (Count_p \leq Sum_p)$;
$Normal(p) \equiv GoodPif(p) \wedge GoodLevel(p) \wedge GoodFok(p) \wedge GoodCount(p)$;
$Leaf(p) \equiv (\forall q \in Neig_p :: (Pif_q \neq C) \Rightarrow (Par_q \neq p))$;
$BLeaf(p) \equiv (Pif_p = B) \Rightarrow (\forall q \in Neig_p :: (Par_q = p) \Rightarrow (Pif_q = F))$;
$BFree(p) \equiv (\forall q \in Neig_p :: Pif_q \neq B)$;
$Broadcast(p) \equiv (Pif_p = C) \wedge Leaf(p) \wedge (Potential_p \neq \emptyset)$;
$ChangeFok(p) \equiv (Pif_p = B) \wedge Normal(p) \wedge (Fok_p \neq Fok_{Par_p})$;
$Feedback(p) \equiv (Pif_p = B) \wedge Normal(p) \wedge BLeaf(p) \wedge Fok_p$;
$Cleaning(p) \equiv (Pif_p = F) \wedge Normal(p) \wedge Leaf(p) \wedge BFree(p)$;
$NewCount(p) \equiv (Pif_p = B) \wedge Normal(p) \wedge (Count_p < Sum_p) \wedge \neg Fok_p$;
$AbnormalB(p) \equiv \neg Normal(p) \wedge (Pif_p = B)$;
$AbnormalF(p) \equiv \neg Normal(p) \wedge (Pif_p = F)$;

Actions:
$B\text{-}action :: Broadcast(p) \longrightarrow Par_p := min_{\prec_p}(Potential_p); L_p := L_{Par_p} + 1; Count_p := 1; Fok_p := false; Pif_p := B;$
$Fok\text{-}action :: ChangeFok(p) \longrightarrow Fok_p := true;$
$F\text{-}action :: Feedback(p) \longrightarrow Pif_p := F;$
$C\text{-}action :: Cleaning(p) \longrightarrow Pif_p := C;$
$Count\text{-}action :: NewCount(p) \longrightarrow Count_p := Sum_p;$
$B\text{-}correction :: AbnormalB(p) \longrightarrow Pif_p := F;$
$F\text{-}correction :: AbnormalF(p) \longrightarrow Pif_p := C;$

Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)
1063-6927/02 $17.00 © 2002 IEEE

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 $GoodLevel$).
3. If p is in the broadcast phase, then if Fok_p is different from that of its parent, then it must be false. Also, if p is in the feedback phase, then Fok_{Par_p} must be true if $Pif_{Par_p} = B$ (Predicate $GoodFok$).
4. If p is in the broadcast phase and still is not involved in the Fok wave, then $Count_p$ must be less than or equal to the sum of the count values of its descendants in $B\text{-}tree_p$ (Predicate $GoodCount$).

Conditions 3 and 4 must also be true if p is the root (Predicates $GoodFok$ and $GoodCount$). A processor conforming to the above rules is called a *normal processor* (Predicate $Normal$). Otherwise, it is called an *abnormal processor*. The correction actions in both Algorithms 1 and 2 ($B\text{-}correction$ and $F\text{-}correction$) are used to correct the abnormal processors. We discuss the correction process in detail in the next section.

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 Section 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. We first show some general properties of the algorithm (Section 4.2). Next, in Section 4.3, we prove that the system will be free of abnormal processor in at most $3 \times L_{Max} + 3$ rounds. Then, we show in Section 4.4 that our algorithm is snap-stabilizing.

4.1 Some Definitions

Definition 3 (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 4 (ParentPath) *For any processor p such that $Pif_p \neq C$, the path $p = p_0, p_1, p_2, \ldots p_k$ is called $ParentPath(p)$ iff the following conditions hold:*
1. $\forall i, 0 \leq i \leq k-1$, $Normal(p_i)$ and $Par_{p_i} = p_{i+1}$.
2. $p_k = r$ or p_k is an abnormal processor.

Definition 5 (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, $q \in Tree(p)$ iff p is an extremity of $ParentPath(q)$.*

Definition 6 (LegalTree) *The tree rooted by r is called the $LegalTree$.*

Definition 7 (Source) *A processor p is called a* source *of $Tree(q)$ iff $p \in Tree(q)$ and for any $q' \in Tree(q)$, $p \neq Par_{q'}$.*

Definition 8 (Normal Configuration) *A configuration γ is called a Normal configuration iff $\forall p : Normal(p)$.*

Definition 9 (Broadcast Configuration) *A configuration γ is called a Broadcast (*B*) configuration iff $Pif_r = B$ and $Fok_r = false$.*

Definition 10 (Start Broadcast Configuration) *A configuration γ is called a Start Broadcast (*SB*) configuration iff $Pif_r = C$.*

Definition 11 (Start Broadcast Normal Configuration) *A configuration γ is called a Start Broadcast Normal (*SBN*) configuration iff γ is both an* SB *and a normal configuration. Note that in γ, $\forall p, Pif_p = C$.*

Definition 12 (End Broadcast Normal Configuration) *A configuration γ is called a End Broadcast Normal (*EBN*) configuration iff γ is normal, $Fok_r = false$, and $\forall p$, $Pif_p = B$. Note that in $\gamma, \forall p, Fok_p = false$.*

Definition 13 (End Feedback Configuration) *A configuration γ is called an End Feedback (*EF*) configuration iff $Pif_r = F$.*

Definition 14 (End Feedback Normal Configuration) *A configuration γ is called an End Feedback Normal (*EFN*) configuration iff γ is both an* EF *and a normal configuration.*

Definition 15 (Good Configuration) *A configuration γ is called a Good Configuration (*GC*) iff $\forall p \notin LegalTree$, $((Pif_p \in \{B, F\}) \wedge (Par_p \in LegalTree)) \Rightarrow GoodCount(p)$.*

Definition 16 (Good LegalTree) *In a Good Configuration, the $LegalTree$ is called* GoodLegalTree *(GLT).*

4.2 General Properties

The following property is an invariant:

Property 1 $((Pif_r = B) \wedge \neg Fok_r) \Rightarrow$
$\forall p,\ (p \in Legaltree \Rightarrow ((Pif_p = B) \wedge ((p \neq r) \Rightarrow (L_p = L_{Par_p} + 1)) \wedge \neg Fok_p \wedge (Count_p \leq Sum_p)))$

Proof. Let p be a processor of the $Legaltree$, and $p = p_0, p_1, \ldots p_k = r$ be $LegalParentPath(p)$. So, $\forall i,\ i \in [0..k],\ p_i \in Legaltree$. Furthermore, $Pif_r = B$, $Fok_r = false$, and $Count_r \leq Sum_r$. Since p_{k-1} is a normal processor of $Legaltree$, $Pif_{p_{k-1}} = B$, $Fok_{p_{k-1}} = false$, and $Count_{p_{k-1}} \leq Sum_r$ (p_{k-1} satisfies $GoodPif(p_{k-1})$, $GoodFok(p_{k-1})$ and $GoodCount(p_{k-1})$). By induction, $Pif_{p_{k-1}} = B$, $Fok_{p_{k-1}} = false$, and $Count_p \leq Sum_r$. □

Property 2 *Let γ be a normal configuration (i.e., no processor is abnormal.) Then, the following properties hold:*
1. $\forall p,\ ((Pif_p \neq C) \Rightarrow (p \in GLT))$
2. $(Pif_r = C) \Rightarrow \forall p,\ (Pif_p = C)$
3. $(Pif_r = F) \Rightarrow \forall p,\ (p \in Legaltree \Rightarrow (Pif_p = F))$
4. $((Pif_r = B) \wedge \neg Fok_r) \Rightarrow$
$(\forall p,\ p \in LegalTree(Count_p \leq \#Subtree(p)))$

Proof. **1.** Let $Pif_p \neq C, p \neq r$. Since p is a normal processor, $ParentPath(p)$ contains at least two processors. Let q be an extremity of $ParentPath(p)$. Since γ is a normal configuration, using Definition 4, $q = r$. So, $ParentPath(p)$ is a $LegalParentPath(p)$ and $p \in GLT$.
2. Let us prove that $\exists p,\ (Pif_p \neq C) \Rightarrow (Pif_r \neq C)$. Let $Pif_p \neq C, p \neq r$. $p \in GLT$ from Case 1. So, $Pif_p \neq C$.
3. Let us prove that $\exists p,\ (p \in Legaltree \wedge (Pif_p \neq F)) \Rightarrow (Pif_r \neq F)$. Let $Pif_p = B$. The case $p = r$ is trivial. So, assume that $p \neq r$. Since p is a normal processor, $ParentPath(p)$ contains at least two processors. Let q be an extremity of $ParentPath(p)$. By Definition 4, q is either an abnormal processor or $q = r$. Since each processor of this path satisfies $GoodPif(p)$, for all processor $q \neq r$ of $ParentPath(p)$, $Pif_q = B$ implies $Pif_{Par_q} = B$. So, $Pif_r = B$ and $Pif_r \neq F$.
4. We prove this by induction on the height of $Subtree(p)$. Let p be a processor such that $p \in LegalTree$ and $height(Subtree(p)) = 1$. So, $Leaf(p)$ is true. Since $Normal(p)$ is true (γ is normal), $GoodCount(p)$ holds and $Count_p \leq 1$. Assume that for all processors p such that $p \in LegalTree$ and $height(Subtree(p)) \leq i$, $Count_p \leq \#Subtree(p)$. Let $q \in LegalTree$ and $height(Subtree(q)) = i + 1$. Then, $\#Subtree(q) = 1 + \sum_{t \in \{u::Par_u = q\}} \#Subtree(t)$. Since q is a normal processor, $Count_q \leq Sum_q$. Furthermore, $Sum_q = 1 + \sum_{t \in \{u::Par_u = p\}} Sum_t$. Since for all $t \in \{u :: Par_u = q\}$, $height(Subtree(t)) \leq i, Count_t \leq Sum_t \leq \#Subtree(t)$, so $Count_q \leq \#Subtree(q)$. □

4.3 Abnormal Processors

In this subsection, we show that in at most $3 \times L_{Max} + 3$ rounds, all abnormal processors will be removed. We first show that every processor p will satisfy $GoodCount(p)$ in at most $L_{Max} + 1$ rounds. Then, starting from this configuration, every processor p will satisfy $Normal(p)$ in at most $2 \times L_{Max} + 2$ rounds.

Every processor eventually holds GoodCount. The following lemma explains why every processor p eventually satisfies $GoodCount(p)$.

Lemma 1 *Let p be a processor such that $GoodCount(p)$ is false at the beginning of a round $\mathcal{R}$. Then, during $\mathcal{R}$, p either executes Action B-$Correction$ or satisfies $GoodCount(p)$.*

Proof. Follows from the definition of a round. □

The next lemma explains the only situations when a processor p may not satisfy $GoodCount(p)$ for the first time. In the following, X_p^i ($Y(p)^i$) denotes the value of X_p (respectively, $Y(p)$) in the global configuration γ_i.

Lemma 2 *Consider a computation step $\gamma^i \mapsto \gamma^{i+1}$. Assume that there exists a processor p such that $GoodCount^i(p)$ is true and $GoodCount^{i+1}(p)$ is false. Then, there exists a processor q such that : $Par_q^i = p$, $L_q^i = L_p + 1$, $Pif_p^i = B$, and $GoodCount^i(q)$ is false. Furthermore, Action B-$correction$ is executed by q during $\gamma^i \mapsto \gamma^{i+1}$.*

Proof. From the algorithm, a descendant q of p disappeared during $\gamma^i \mapsto \gamma^{i+1}$. So, q executed its B-$correction$. Since the value of $Count_q$ was included in Sum_p^i, we have $Par_q^i = p$, $L_q^i = L_p + 1$, and $Pif_p^i = B$. So, q executed its B-$correction$ because $GoodCount^i(q)$ was false. □

Property 3 *After at most $L_{Max} + 1$ rounds, $\forall p$: $GoodCount(p)$ is true forever.*

Proof. From Lemmas 1 and 2, the level of every processor p such that $\neg GoodCount(p)$ strictly decreases. So, the property holds since $GoodCount(p)$ cannot be negative. □

Every processor is eventually Normal. Let us assume that in every configuration, $GoodCount(p)$ is true for each processor p.

Lemma 3 *Let p be a processor and $\gamma^i \mapsto \gamma^{i+1}$ be a computation step such that $Normal^i(p)$ is false. Let $q = Par_p^i$. If $Normal^{i+1}(p)$ holds, then either p executes a correction action or q executes Fok-$action$ during $\gamma^i \mapsto \gamma^{i+1}$.*

Proof. First, note that the correction actions are the only enabled actions in γ^i on p if $Normal^i(p)$ does not hold. Let $q = Par_p^i$ and consider the three following cases.
1. $Pif_q^i = C$: since $Leaf^i(q)$ does not hold, q cannot execute any action. Then, $Pif_q^{i+1} = C$. So, if p does not

execute any action, then $Normal^{i+1}(p)$ does not hold.
2. $Pif_q^i = F$: The only action that q can execute is $F\text{-}correction$. Then, $Pif_q^{i+1} = C$. So, if p does not execute any action, then $Normal^{i+1}(p)$ does not hold.
3. $Pif_q^i = B$: Then $Pif_p^i = B$ and either $GoodLevel^i(p)$ or $\neg GoodFok^i(p)$. In the first case, q cannot change its level. In the second case, if p does not execute any action, then the only way $Normal(p)^{i+1}$ can be true is if q executes $Fok\text{-}action$ during $\gamma^i \mapsto \gamma^{i+1}$. □

Lemma 4 *Let p be an abnormal processor in Configuration γ_i. Then, p is a normal processor in at least one configuration during the next two rounds.*

Proof. There are two cases:
1. $Pif_p^i = F$. In at most one round, p executes $F\text{-}correction$ and becomes a normal processor.
2. $Pif_p^i = B$. In at most one round, p executes $B\text{-}correction$. So, Pif_p becomes equal to F. If p is not already a normal processor, then p now will become one by Case 1. □

Lemma 5 *Consider a processor p and a computation step $\gamma^i \mapsto \gamma^{i+1}$ such that $Normal^{i+1}(p)$ is false and $Normal^i(p)$ is true. Let $q = Par_p^{i+1}$. Then, $Normal^i(q)$ is false and q executes a correction action during $\gamma^i \mapsto \gamma^{i+1}$. Furthermore, $L_p^{i+1} = L_q^{i+1} + 1$.*

Proof. Since $Normal^i(p)$ holds, neither q nor p can change its level. So, $GoodLevel^{i+1}(p)$ holds. Assume that $GoodPif^{i+1}(p)$ does not hold. Note that p cannot execute any action leading to this configuration. So, q executes an action. Furthermore, q executes a correction action because no other action can be executed by q. Since $GoodLevel^{i+1}(p)$ is true, $Pif_p^{i+1} = B$ implies $L_p^{i+1} = L_q^{i+1} + 1$. □

Corollary 1 *Let S^{ro} be the set of abnormal processors at the beginning of the round ro. Let lev^{ro} be the minimal level of processors in S^{ro}. Then, $lev^{ro} \leq lev^{ro+1} < lev^{ro+2}$.*

Proof. This is a direct consequence of Lemmas 3 and 5. Note that two rounds may be necessary to execute $B\text{-}correction$ and $F\text{-}correction$ (see Lemma 4). □

The following result follows directly from Corollary 1:

Corollary 2 *Starting from a configuration where $GoodCount(p)$ is true for each processor p, in at most $2 \times L_{Max} + 2$ rounds, every processor becomes normal.*

The following theorem directly follows from Property 3 and Corollary 2:

Theorem 1 *In at most $3 \times L_{Max} + 3$ rounds, every processor will become normal.*

4.4 Proof of Snap-Stabilisation

Theorem 2 *Let γ^i be a global configuration such that $GLT^i \neq \emptyset$. One of the following propositions holds:*
1. *If $Pif_r^i = F$, the network reaches an SB Configuration in at most $4 \times L_{Max} + 4$ rounds.*
2. *If $Pif_r^i = B$ and $Fok_r^i = true$, the network reaches an EF Configuration in at most $5 \times L_{Max} + 4$ rounds.*
3. *If $Pif_r^i = B$ and $Fok_r^i = false$, the network reaches an EBN Configuration in at most $5 \times L_{Max} + 4$ rounds.*

Proof. Consider the three following cases:
1. $Pif_r^i = F$. Let s be any source of GLT^i. Then, $Pif_s^i = F$ and $Normal^i(s)$. If s cannot execute any action in the next step, then either $Leaf^i(s)$ or $BFree^i(s)$ is false. In both cases, there exists an abnormal processor. In the worst case, all the abnormal processors become normal before s may execute a $C\text{-}action$. So, it takes at most $4 \times L_{Max} + 4$ rounds to reach an SB Configuration.
2. $Pif_r^i = B$ and $Fok_r^i = true$. First, L_{Max} rounds are necessary to propagate the Fok value from the root to the source of GLT. Then, by applying the same reasoning as before (it is then an $F\text{-}action$), at most $5 \times L_{Max} + 4$ rounds are necessary to reach an EF Configuration.
3. $Pif_r^i = B$ and $Fok_r^i = false$. First, note that if every processor is in GLT, then the property holds. By Definition 16, Fok_r cannot be true unless every processor is in GLT. Let p be a processor such that $N_p \cap GLT^i \neq \emptyset$. As before, if it is not possible to add p to GLT, then some abnormal processors exist in the network. Then, following the same reasoning as before, at most $5 \times L_{Max} + 4$ rounds are necessary to reach an EBN Configuration. □

Theorem 3 *Starting from any configuration, the protocol creates the GLT in at most $8 \times L_{Max} + 7$ rounds.*

Proof. First, note that if $Pif_r = C$, then r just needs to wait for the removal of the abnormal processors. Once every processor is normal, r executes an action. Assume that $LegalTree$ is not the GLT. So, there exists an abnormal processor in the network. The GLT is obtained just after the abnormal processors are removed. Finally, assume that $LegalTree$ is already the GLT. Then From Theorem 2, $Pif_r = C$ after at most $8 \times L_{Max} + 7$ rounds. □

5 Complexity Analysis

In this section, we consider the time required to complete a PIF cycle starting from an SBN configuration.

Theorem 4 *Starting from an SBN configuration, the protocol executes a PIF cycle in at most $5 \times h + 5$ rounds, where h is the height of the constructed tree during this cycle. Note that $h \in \Omega(diameter)$ and that h is bounded by the length of the longest elementary chordless path in the network.*

Proof. Let us first recall a definition. A path $p_0, p_1, \ldots, p_k = r$ is an elementary chordless pass iff for any i, j $i < j$, $p_i \neq p_j$ (elementary path) and p_i, p_j are linked in the network iff $j = i + 1$ (chordless path). Macro $Potential_p$ implies that our algorithm creates only chordless ParentPaths. Assume the contradiction. Let $p_0, \ldots, p_k$ be a parent path created by the algorithm, and $p_i p_j$ is a chord of this path. Without any loss of generality, assume that $i \leq j - 2$. Since the path is the result of an execution of the algorithm, when p_i chooses its parent, p_i uses $Potential_{p_i}$. So, $L_{p_j} < L_{p_{i+1}}$, and p_{i+1} cannot be in $Potential_{p_i}$. Thus, p_{i+1} cannot be taken as a parent by p_i. A contradiction. Since there are no abnormal processors in the network, $h + 1$ rounds are necessary to reach an EBN configuration. Then, h more rounds are necessary again to obtain the right Count at the root. Next, $Fok_r = true$ and h new rounds are required to propagate Fok to the leaf of the constructed tree. Finally, $h + 1$ rounds are executed to set Pif_r to F, followed by $h + 1$ extra rounds to reach the SBN configuration. □

6 Conclusions

We presented a snap-stabilizing PIF algorithm on an arbitrary network. The algorithm does not use a preconstructed 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 can be used to design a *universal transformer* [13] to provide a snap-stabilizing version of a wide class of 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.

[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] 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.

[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.

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

[8] A. Bui, A. Datta, F. Petit, and V. Villain. Space optimal PIF algorithm: Self-stabilizing with no extra space. In *IPCCC'99, IEEE International Performance, Computing, and Communications Conference*, pages 20–26. IEEE Computer Society Press, 1999.

[9] A. Bui, A. 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.

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

[11] N. Chen, H. Yu, and S. Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39:147–151, 1991.

[12] 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.

[13] A. Cournier, A. Datta, F. Petit, and V. Villain. Snap-stabilizing systems. Technical Report RR01-10, LaRIA, University of Picardie Jules Verne, 2001.

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

[15] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[16] 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.

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

[18] H. Kruijer. Self-stabilization (in spite of distributed control) in tree-structured systems. *Information Processing Letters*, 8:91–95, 1979.

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

[20] M. Raynal and J. Helary. *Synchronization and Control of Distributed Systems and Programs*. John Wiley and Sons, Chichester, UK, 1990.

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

[22] G. Tel. *Introduction to distributed algorithms*. Cambridge University Press, 1994.

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

Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)
1063-6927/02 $17.00 © 2002 IEEE