Snap-Stabilizing PIF on Non-Oriented Trees and Message Passing Model

Florence Levé, Khaled Mohamed, and Vincent Villain

Laboratoire MIS, Université de Picardie, 33 Rue St Leu,
80039 Amiens Cedex 01, France.
{florence.leve,khaled.mohamed,vincent.villain}@u-picardie.fr

Abstract. Starting from any configuration, a *snap-stabilizing protocol* guarantees that the system always behaves according to its specification while a *self-stabilizing protocol* only guarantees that the system will behave according to its specification in a finite time. So, a snap-stabilizing protocol is a time optimal self-stabilizing protocol (because it stabilizes in 0 rounds). That property is very suitable in the case of systems that are prone to transient faults. There exist a lot of approaches of the concept of self-stabilization, but to our knowledge, snap-stabilization is the only variant of self-stabilization which has been proved power equivalent to self-stabilization in the context of the state model (a locally shared memory model) and for non anonymous systems. So the problem of the existence of snap-stabilizing solutions in the message passing model is a very crucial question from a practical point of view. In this paper, we present the first snap-stabilizing propagation of information with feedback (PIF) protocol for non-oriented trees in the message passing model. Moreover using slow and fast timers, the round complexity of our algorithm is in $\theta(h \times k)$ and $\theta((h \times k)+k^2)$, respectively, where h is the height of the tree and k is the maximal capacity of the channels. We conjecture that our algorithm is optimal.

1 Introduction

The concept of Propagation of Information with Feedback (PIF) has been introduced by Chang [7] and Segall [21]. 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 [9].

Self-stabilization has been introduced by Dijsktra in 1974 [19]. A distributed algorithm is self-stabilizing if, starting from any arbitrary global state, the system is able to recover itself in finite time. This property is crucial when considering systems after any faulty behavior (even any byzantine behavior). In that case, the resulting configurations (unexpected messages and memory states) can be arbitrary and self stabilizing algorithms eventually recover without any external action. So it is one of the most versatile techniques to handle transient faults

arising in distributed systems. Recently, snap-stabilization has been introduced by Bui *et al.* [4]. A distributed algorithm is snap-stabilizing if starting from any arbitrary global state the system always satisfies the specification. In other words, a snap-stabilizing algorithm is a self-stabilizing algorithm that stabilizes in 0 rounds, i.e., it is optimal in terms of the stabilization time. For example, after some transient faults, when a user starts a self-stabilizing PIF algorithm he is not sure to receive the right feedback of all the processors at the first attempt, and it is generally the same for all the other attempts until the algorithm stabilizes. Moreover the number of attempts that fail can be not bounded depending on the algorithm. On the contrary, if the algorithm is snap-stabilizing, the first attempt is the *right one*.

There exist a lot of approaches of the concept of self-stabilization, some of them try to overcome some of its drawbacks like a high complexity [17], some of them try to make it stronger [18]. Snap-stabilization belongs to the second family and enhances the safety of the system since the stabilization time is nul. To our knowledge, snap-stabilization is the only variant of self-stabilization which has been proved power equivalent to self-stabilization in the context of the state model (a locally shared memory model) and for non anonymous systems, meaning that each problem that admits a self-stabilizing solution also admits a snap-stabilizing solution and reciprocally [11]. So the existence of snap-stabilizing solutions in the message passing model is a very crucial problem from a practical point of view.

Related work. Several snap-stabilizing PIF protocols have been proposed for oriented trees [4, 20, 6], for non-oriented trees [5, 10, 12, 6], for full-connected networks [16], and for general networks [8, 3, 14]. In [11, 13, 15] snap-stabilizing PIF protocols are the key tools of the transformation of protocols into snap-stabilizing versions. But all the literature above is written in the state model.

To the best of our knowledge there only exist two snap-stabilizing protocols in the message passing model. A snap-stabilizing propagation of information with feedback (PIF) protocol for full-connected networks has been presented in [16]. Another one for oriented trees has been evoked in [20] but no protocol nor proof are provided.

Contributions. In this paper, we present the first snap-stabilizing PIF algorithm for non-oriented trees in the message-passing model. Following the impossibility result in [16] we consider that the capacity of the channels is bounded. Moreover in order to tolerate message losses after the occurence of transient faults our algorithm uses timers that regularly send duplicate messages. With slow timers, we show that the round complexity of a complete execution is in $\theta(h \times k)$ where h and k are the height of the tree and the maximal capacity of the channels, respectively. With fast timers, the round complexity only gets k^2 as an additionnal term and is in $\theta((h \times k) + k^2)$. We conjecture that our algorithm is optimal in terms of round complexity. That result is coherent with the round complexity in [5] and [6] since they achieve $\theta(h)$ in the state model.

Outline of the paper. The paper is organized as follows. In Section 2 we present the model assumed in this paper. We then present the snap-stabilizing PIF algorithm and its proof in Section 3 and discuss the complexity in Section 4. We then conclude in the last section.

2 Preliminaries

Notations. We consider a network as an undirected connected graph $G = (V, E)$ where V is a finite set of nodes (or *processors*) ($|V| = n$) and E is the set of *bidirectional asynchronous communication links*. A bidirectional communication link $\{p, q\}$ exists iff p and q are neighbors, in this case p and q can communicate together by sending messages through the link. This link can be viewed as two *channels* (p, q) and (q, p), one by direction. The capacity of the channels is bounded, otherwise no deterministic snap-stabilizing solution is available [16]. To simplify the presentation, we assume that the bound is the same for every channel and is denoted by k. Channels are not reliable so messages can be lost but they are *fair*, i.e., if a processor sends infinitely many messages through a channel, then the channel will deliver infinitely many of them. Messages can be lost when the channel has a faulty behavior or is full. When they are not lost, messages transmitted through a channel are received in a finite but not bounded time, moreover they arrive in the order they have been sent (FIFO). Every processor p can distinguish and number all its channels from 0 to $\delta - 1$, where δ is the number of neighbors of p. For sake of simplicity, we sometimes refer to a link $\{p, q\}$ (or a channel (p, q)) of a processor p by the label q instead of its local number. We consider networks which are *tree structured*, so $|E| = n - 1$. Our algorithm will not use any identity for the processors except the one called the *root* or r. Any other processor p is called an *internal* processor if it has at least two neighbors, and a *leaf* processor otherwise.

We will call the neighbor of a processor p ($p \neq r$) which is on the path from r to p the *topological parent* of p. We will call any of the other neighbors of p a *topological child*. In non-oriented trees, p does not know which of its neighbors is its topological parent.

Programs. In our model, protocols are *semi-uniform*, i.e., according to δ, each processor executes the same program except r. So, aside from r, we distinguish between the case of internal processors ($\delta > 1$) and leaf processors ($\delta = 1$). We consider the message-passing model of computation. The message receptions are sequentially taken into account by the processors and the set of actions associated to a message reception is atomically executed.

To compute the time complexity, we use the notion of *round*. Since in asynchronous systems, the local execution time is considered as null, the definition of a *round* captures the execution rate of the slowest messages in any computation.

Definition 1 (Round). *Given an execution e of a protocol P, the* first round *of e (let us call it e') is the minimal prefix of e containing the reception or the*

loss of every message sent or already in a channel 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.*

We assume that during a round the timers of every processor involved in the execution are activated at least once.

PIF. PIF is a well-known problem, so we simply specify the problem as follows:

Specification 1 (PIF) *An algorithm is a PIF algorithm if it satisfies the two following conditions:*

[PIF1] *r initiates a PIF by broadcasting a message m,*
[PIF2] *after the initialization, PIF terminates at r and when that happens, all processors have acknowledged the receipt of m.*

Remark 1. In practice, to prove that a PIF protocol is snap-stabilizing we have to show that every execution of the algorithm satisfies these two conditions: (i) if r has a message m to broadcast, it will do it in a finite time, and (ii) starting from any configuration where r broadcasts m, the system satisfies Specification 1.

3 Snap-Stabilizing PIF Algorithm

3.1 Algorithm Description

We call our algorithm *PIF*, the formal description is given in Algorithms 1, 2, 3, and 4. The main idea follows that of [4, 6] while cleaning the channels follows that of [16] based itself on a sequence of integers as in [2, 1]. Roughly speaking, our algorithm is based on the sending of broadcast messages with timestamp (Messages *("B",X)*) in order to stabilize the channels. An internal processor must wait for a broadcast message with Timestamp $M - 2$ before it begins to propagate the broadcast. The value M is discussed below in paragraph *Any Initial Configuration.* In order to avoid wrong feedbacks, a processor which has received the feedback (Message *("F")*) of all its children must wait for an application (Message *("B",$M-1$)*) from its parent before it sends it a feedback. We describe our algorithm more precisely in the two following paragraphs.

Safe Initial Configuration. Starting from a safe configuration, *i.e.,* no processor is involved in any execution of *PIF* (so Boolean *End* is $True$) and every channel is empty, r sends *("B",0)* to all its topological children, see Actions *Spontaneously* and *Timer[C]=0* in Algorithm 2 and Macro *InitBroadcast()* in Algorithm 1.

At the reception of *("B",0)* from channel C, a neighbor p of r sets $parent_p$ to C, $S_p[C]$ to 0, and sends *("ACK",0)* to r by C, see Action *At the reception of ("B",X) from C* in Algorithms 4 or 3. At the reception of *("ACK",0)* from channel C, r sets $S_r[C]$ to 1 and sends *("B",1)* to C, see Action *At the reception of ("ACK",X) from C* in Algorithm 2. This exchange of messages goes on on every channel of r. When for some C, $S_r[C]$ reaches $M - 2$, r sends *("B",M-2)* to C. When receiving this message, the neighbor p at the end of C sends

("ACK",M-2) to r, sets $S_p[C]$ to $M-2$, and starts broadcasting *("B",0)* to all its topological children. At the reception of *("ACK",M-2)*, r sets $S_r[C]$ to $M-1$ and sends *("B",M-1)* to C. When p receives this message, it sets $S_p[C]$ to $M-1$ meaning that r is now waiting for the feedback message *("F")* of p. The broadcast goes on along the paths from r to the leaves. When a leaf has received the sequence of *("B",0)*, *("B",1)*,..., *("B",M-1)* from its unique channel 0, it sends back *("F")*, see *At the reception of ("B",X) from C* in Algorithm 3. After they send their sequence of *("B",0)*, *("B",1)*,..., *("B",M-1)*, the internal processors wait for a *("F")* from their topological children. Once an internal processor p has received a *("F")* from all its children, $S_p[C]$ equals M for all its children, so that p satisfies the predicate $EndWaitFeedback$ and it can sends *("F")* to its topological parent. We can remark that $S_p[parent_p]$ can be equal to $M-2$ only because the system is asynchronous. In this case p waits for the reception of *("B",M-1)* from its parent before p sends *("F")*. Finally, the *("F")* messages go up to r and the execution terminates.

Any Initial Configuration. Starting from any configuration, the channels can contain some residual messages (messages already in the channels at the initial configuration). Since $M = (2k+1)+2$ where k is the channel capacity, the sequence of *("B",0)*, *("B",1)*,..., *("B",M-2)* ensures that when the sender receives the last acknowledgment *("ACK",M-2)* the receiver has received at least *("B",M-2)* and that *("ACK",M-2)* is really an acknowledgment to this message.

But cleaning the channels is not enough to ensure a good behavior of the algorithm, because residual messages may have bad consequences when proceeding by a processor. The role of messages *("Broadcast?")*, *("B:No")*, and *("Withdraw")* is to limit the effects of a possible dysfunction. A broadcasting internal processor p regularly sends *("Broadcast?")* to its neighbor $parent_p$ to be sure that $parent_p$ is really broadcasting a message to p. If $parent_p$ is still broadcasting to p then $parent_p$ does not answer directly to the question since it will eventually send a broadcast message to p. Otherwise, *parent* sends a *("B:No")* message to p. When a processor p, which is not involved in any *PIF* execution (*i.e.,* End_p is $True$), receives a *("B",M-2)* message from a neighbor q (*i.e.,* q is waiting for a feedback), it sends a *("Withdraw")* message to q, because the broadcast is not valid. These messages allow to remove abnormal broadcasts, *i.e.,* broadcasts the source of which is not r.

3.2 Proof of Snap-Stabilization

To simplify the proofs, we will reduce the tree to a chain where r is the processor at the top of the chain and the unique leaf is the processor at the bottom of the chain. The validity of our results on trees is simply ensured by the synchronization of the ("F") sending, driven by the predicate $EndWaitFeedback$.

We now present several definitions before we prove our algorithm.

Definition 2 (Residual Message). *A message which is already in a channel in the initial configuration is called a* residual *message.*

Algorithm 1: Environment

Constant :
M // M=(2k+1)+2 where k is an integer bounding the channel capacity
α: positive integer // waiting time value of the timer
δ: positive integer // number of neighbors of the processor

Messages :
("B",X) where $X \in \{0, \ldots, M-1\}$
("ACK",X) where $X \in \{0, \ldots, M-2\}$
("F")
("F_ACK")
("Broadcast?")
("Withdraw")
("B:No")

Variables :
C, parent, i: channel number
$S[0 \ldots \delta - 1]$: array of values in $\{0, \ldots, M\}$
End: boolean
Prev: in $\{0, \ldots, M\}$

Predicates:
$Broadcast[C] \equiv (S[C] \leq M-2)$ // r
$Broadcast[C] \equiv (S[parent] \in \{M-2, M-1\}$ and $S[C] \leq M-2)$ // internal proc.
$WaitFeedback[C] \equiv (S[C] = M-1)$ // r
$WaitFeedback[C] \equiv (S[parent] \in \{M-2, M-1\}$ and $S[C] = M-1)$ // internal proc.
$EndWaitFeedback \equiv (\forall i \in \{0, \ldots, \delta-1\}, S[i] = M)$ // r
$EndWaitFeedback \equiv (\forall i \in \{0, \ldots, \delta-1\} \setminus \{parent\}, S[i] = M$ and $S[parent] = M-1)$
// internal proc.
$EndWaitFeedback \equiv (S[0] = M-1)$ // leaves
$Error \equiv (S[parent] \notin \{M-2, M-1\}$ and $\exists C \in \{0, \ldots, \delta-1\} \setminus \{parent\} : S[C] \neq 0)$
// internal proc.

Macros :
$ExecEnd() : (\forall i \in \{0, \ldots, \delta-1\}(S[i] \leftarrow 0); End \leftarrow True)$ // leaves and internal proc.
$InitBroadcast() : (\forall i \in \{0, \ldots, \delta-1\}(S[i] \leftarrow 0; Timer[i] \leftarrow 0))$ // r
$InitBroadcast(C) : (\forall i \in \{0, \ldots, \delta-1\} \setminus \{C\}(S[i] \leftarrow 0; Timer[i] \leftarrow 0))$ // internal proc.

Algorithm 2: r code

```
• Spontaneously
if End or EndWaitFeedback then
    end ← False ; InitBroadcast() ;
end
• At the reception of ("ACK",X) from C
if ¬(End) then
    if S[C]=X then
        S[C] ← S[C]+1;
        if Broadcast[C] or WaitFeedback[C] then
            Timer[C] ← 0;                                   // Send (''B'', S[C]) to C
        end
    end
end
• At the reception of ("F") from C
if ¬(Broadcast[C]) then
    Send("F_ACK") to C;
    if ¬(End) and WaitFeedback[C] then
        S[C] ← M;
        if EndWaitFeedback then
            ExecEnd();
        end
    end
end
• At the reception of ("Broadcast?") from C
if End or EndWaitFeedback then
    Send("B:No") to C;
end
• At the reception of ("Withdraw") from C
if ¬(Broadcast[C]) and ¬(End) then
    ExecEnd();
end
• Timer[C]=0
if ¬(End) then
    if Broadcast[C] or WaitFeedback[C] then
        Send("B",S[C]) to C;
    end
    Timer[C] ← α;
end
```

Algorithm 3: Leaves code

```
• At the reception of ("B",X) from C                                   // C=0
if End then
    if X=M-1 then
        Send("Withdraw") to 0 ;
    else
        if X < M-1 then
            End ← False ;
        end
    end
else
    Prev ←S[0] ;
    S[0] ← X ;
    if S[0] > M-1 or S[0] < Prev then
        ExecEnd() ;
    else
        if S[0] < M-1 then
            Send ("ACK", X) to 0 ;
        else // S[0]=M-1
            Send ("F") to 0 ;
        end
    end
end
• At the reception of ("F_ACK") from C                                 // C=0
if ¬(End) then
    ExecEnd() ;
end
• At the reception of ("Broadcast?") from C                            // C=0
Send ("B:No") to 0 ;
```

Definition 3 (Working Message). *A message* m *received by* p *is a* working *message if* p *executes at least one state change at the reception of* m.

Definition 4 (Real Broadcast). *Let* p *and* q *be two neighboring processors. A* ("B",X) *message received by* p *from channel* (q, p) *is a* real broadcast *if it has been sent by* q.

Definition 5 (Real Acknowledgment). *Let* p *and* q *be two neighboring processors. A* ("ACK",X) *message received by* p *from channel* (q, p) *is a* real acknowledgment *if it has been sent by* q *at the reception of a* real broadcast ("B",X) *sent by* p.

Definition 6 (Real Feedback). *Let* p *and* q *be two neighboring processors. A* ("F") *message received by* p *from channel* (q, p) *is a* real feedback *if it has been sent by* q *at the reception of a* real broadcast ("B",M-1) *sent by* p.

Definition 7 (Right Feedback). *Let* p *and* q *be two neighboring processors. A* ("F") *message received by* p *from channel* (q, p) *is a* right feedback *if it is a* real feedback *sent by* q *and* q *satisfies one of the two following conditions:*

1. q *is a leaf,*
2. q *is not a leaf. Let* q' *be the topological child of* q. *The last* ("F") *message* q *has accepted is a* right feedback *from* q'.

Roughly speaking a ("F") *message is a* right feedback *if this message is a* real feedback *which has originally been generated by the leaf.*

We define an *abnormal root* as an internal processor which is broadcasting while it must not. More formally:

Algorithm 4: Internal Processors code

```
• At the reception of ("B",X) from C
if End then
    if X=M-1 then
        Send("Withdraw") to C ;
    else
        if X < M-1 then
            End ← False ; parent ← C ;
        end
    end
end
if ¬(End) and C=parent then
    Prev ← S[C] ; S[C] ← X ;
    if S[C] > M − 1 or S[C] < Prev or (S[C] = M − 1 and Prev < M − 2) or Error then
        ExecEnd();
    else
        if S[C] < M − 1 then
            Send ("ACK",X) to C ;
            Timer[C] ← α ;                                  // Avoid to send (''Broadcast?'') to parent
            if S[C] = M − 2 and S[C] ≠ Prev then
                InitBroadcast(C) ;
            end
        else // S[C]=M-1
            if EndWaitFeedback then
                Send("F") to C ;
            end
        end
    end
end
• At the reception of ("ACK",X) from C ∈ {0, . . . , δ − 1} \ {parent}
if ¬(End) then
    if S[C]=X then
        S[C] ← S[C]+1 ;
        if Broadcast[C] or WaitFeedback[C] then
            Timer[C] ← 0 ;                                  // Send (''B'', S[C]) to C
        end
    end
end
• At the reception of ("F") from C ∈ {0, . . . , δ − 1} \ {parent}
if ¬(Broadcast[C]) then
    Send("F_ACK") to C ;
    if ¬(End) and WaitFeedback[C] then
        S[C] ← M ;
        if EndWaitFeedback then
            Send("F") to parent ;
        end
    end
end
• At the reception of ("F_ACK") or ("B:No") from parent
if ¬(End) then
    ExecEnd();
end
• At the reception of ("Broadcast?') from C
if ¬(Broadcast[C]) and ¬(WaitFeedback[C]) then
    Send("B:No") to C ;
end
if C=parent then
    ExecEnd() ;
end
• At the reception of ("Withdraw") from C
if ¬(Broadcast[C]) and ¬(End) then
    ExecEnd();
end
• Timer[C]=0
if ¬(End) then
    if Error then
        ExecEnd() ;
    else
        if Broadcast[C] or WaitFeedback[C] then
            if C ≠ parent then
                Send("B",S[C]) to C ;
            else
                Send("Broadcast?") to C ;
            end
        end
    end
    Timer[C] ← α;
end
```

Definition 8 (Abnormal Root). *Let p be an internal processor. Processor p is an* abnormal root *if all the following conditions are satisfied:*

1. $\neg End_p$,
2. $Broadcast_p \vee WaitFeedback_p$,
3. $Error_q \vee End_q \vee (\neg End_q \wedge (parent_q = p)) \vee (\neg End_q \wedge (parent_q \neq p) \wedge (S_q[p] = M))$ *where* $q = parent_p$.

Processor p is a top abnormal root *or a* bottom abnormal root *if q is the topological parent of p or a topological child of p, respectively.*

Definition 9 (Down and Up Broadcast). *Let p be either r or a* top abnormal root *(respectively a* bottom abnormal root*). If p is sending* ("B",X) *messages, this broadcast is called a* down broadcast *(respectively an* up broadcast*).*

Definition 10 (Up-Free Configuration). *A configuration is called an* up-free *configuration if it does not contain any* bottom abnormal root *nor any* up broadcast *message.*

We will often assume in the proofs that a processor which starts or has started a broadcast is never disrupted by its parent (that is trivially satisfied by r since it has no parent). We call such a processor a *stop-free processor*.

Definition 11 (Stop-Free Processor). *Let p be a processor, p is a* stop-free processor *if one of the two following conditions is satisfied:*

1. *p is* r,
2. *p is not* r *and all the following conditions are satisfied:*
   - *p never receives from its parent a message* ("B",X) *followed by* ("B",Y) *with $X > Y$,*
   - *p never receives from its parent any message* ("F_ACK"), ("B:No"), ("Withdraw").

In order not to overload the proofs, since after one round, every processor does not satisfy *Error* forever, we now assume that no processor will satisfy *Error* forever.

Lemma 1 (Abnormal Root Life). *Let p be an* abnormal root*. Processor p cannot remain an* abnormal root *forever.*

Proof. Assume, by the contradiction, that p remains an *abnormal root* forever. Let q be the neighbor of p such that $parent_p = q$. Then p sends *("Broadcast?")* infinitely often to q. By fairness of (p, q), q will receive *("Broadcast?")* infinitely often. Since p is an *abnormal root* forever, when q receives *("Broadcast?")* from (p, q), q follows one of these three cases:

- End_q is satisfied,
- q is not r and $(\neg End_q \wedge (parent_q = p))$ is satisfied,
- q is not r and $(\neg End_q \wedge (parent_q \neq p) \wedge (S_q[p] = M))$ is satisfied.

In all cases q sends *("B:No")* to p. So q sends *("B:No")* infinitely often to p and by link fairness, p receives *("B:No")* infinitely often. Since at the reception of *("B:No")* p executes $ExecEnd()$, p will stop being an *abnormal root* in a finite time. A contradiction. □

The following lemma can be easily proved by induction on the distance to the leaves:

Lemma 2 (Bottom Abnormal Root Appearance). *Let p be an internal processor. Processor p cannot become a* bottom abnormal root *infinitely often.*

From Lemmas 1 and 2 we can easily deduce the following lemma:

Lemma 3 (No More Bottom Abnormal Root). *The number of* bottom abnormal roots *is nul in a finite time.*

Since *up broadcast* messages can be only generated by *bottom abnormal roots*, we can easily deduce the following corollary from Lemma 3.

Corollary 1 (No More Up Broadcast). *The system does not contain any message of an* up broadcast *in a finite time.*

We first show that starting from an *up-free* configuration our algorithm always satisfies Specification 1 (from Lemma 4 to Theorem 1). We then prove that the result still holds from any initial configuration so even if this configuration contains *bottom abnormal roots*.

Lemma 4 (Broadcast Progress). *Let p and q be two neighboring processors such that p sends* ("B",X) *($X \in \{0, \ldots, M-2\}$) to q from an* up-free *configuration. If p is stop-free it will eventually receive* ("ACK",X) *by channel* (q, p).

Proof. Assume that p never receives any message *("ACK",X)* from q. So p sends infinitely many *("B",X)* to q. Since (p, q) is fair, q receives infinitely many *("B",X)*. The configuration is *up-free*, so $parent_q$ cannot be set to another neighbor than p. Depending of the initial state of q, q can execute $ExecEnd()$ at the reception of the first *("B",X)* from p. But at each new reception q sends *("ACK",X)* to p. So q sends infinitely many *("ACK",X)* and by fairness of (q, p) p will eventually receive infinitely many *("ACK",X)*. A contradiction. □

The following lemma states that if a processor sends all its *("B",X)* messages to a neighbor, then it is ensured that its neighbor has received *("B",M - 2)*.

Lemma 5 (Snap Local Broadcast). *Let p and q be two neighboring processors such that p starts broadcasting to q (it sends* ("B",0) *to q) from an* up-free *configuration. If p is stop-free it will eventually send* ("B",M - 2) *to q and, for this message, it will receive a* real acknowledgment *from q.*

Proof. From our algorithm, since p is stop-free, p increments $S_p[q]$ at the reception of *("ACK",X)* from q, then sends *("B",$S_p[q]$)* to q and goes on until it receives *("ACK",M-2)* from channel (q, p). Thus from Lemma 4, if the processor

p starts broadcasting to q, we know that p will eventually receive a message *("ACK",M-2)* by channel (q, p). Since $M - 2 = 2k + 1$ where k is the channel capacity, this message cannot be a residual channel message, it has necessarily been generated by q. □

The following lemma can be formally proved by induction on the distance to the farthest leaf.

Lemma 6 (Partial Snap Up-Free PIF). *Let p be a processor such that p is not a leaf and p starts a* down broadcast *(it sends* ("B",0) *to its topological child) from an* up-free *configuration. If p is stop-free it will eventually receive a unique working* ("F") *message and this message is a* right feedback*.*

Since by Definition 11 r is stop-free, r satisfies Lemma 6 and *[PIF2]* of Specification 1 holds. Moreover, if r is already involved in Algorithm *PIF* in the initial *up-free* configuration, it is easy to show with a reasoning similar to that of the proof of Lemma 6 that r will be able to initiate a complete PIF in a finite time, so the following result holds:

Lemma 7 (Up-Free PIF1). *Starting from an* up-free *configuration, Algorithm* PIF *satisfies* [PIF1] *of Specification 1.*

The following theorem is a corollary of Lemmas 6 and 7:

Theorem 1 (Snap Up-Free PIF). *Starting from an* up-free *configuration, Algorithm* PIF *is snap-stabilizing.*

We now generalize Lemma 6 and Theorem 1 to any initial configuration.

Lemma 8 (Partial Snap PIF). *Let p be a processor such that p is not a leaf and p starts a* down broadcast *from any initial configuration. If p is stop-free it will eventually receive a unique working* ("F") *message and this message is a* right feedback*.*

Proof. We know from Lemma 6 that if a non-leaf processor p starts a *down broadcast* from an *up-free* configuration, and if p is stop-free, it will eventually accept a unique *("F")* message and this message is a *right feedback*.

Now consider the case when there exist abnormal roots in the configuration. A top abnormal root does not cause any problem to the execution of the algorithm since messages ("B",X) coming from its parent are working messages (when its state is different from X). Assume that q and q' are two neighboring processors, such that q starts a down broadcast towards q' and q' is a bottom abnormal root (w.l.o.g. we can suppose there is no bottom abnormal root on the topological path from the root to q'). No message can prevent q from broadcasting because:

- Since q' is not the parent of q, if q receives *("B",No)* from q', it does not take it into account.
- Since $S_q[q'] < M - 1$, if q receives *("Withdraw',)* from q', it does not take it into account.

– Since q satisfies $Broadcast$, if q receives *("F")* from q', it does nothing.
– Since the parent of q is stop-free, it does not send *("F_ACK")* to q until it receives F from q.

From Lemma 3, we know that the number of bottom abnormal roots is nul in a finite time. Thus the broadcast will go on towards the leaves and, from Lemma 6, p will eventually receive a unique working *("F")* message and this message is a *right feedback*. □

Since r is stop-free, r satisfies Lemma 8 and *[PIF2]* of Specification 1 holds. Moreover, as previously, if r is already involved in Algorithm *PIF* in the initial configuration, it is easy to show that r will be able to initiate a complete PIF in a finite time, so the following result holds:

Lemma 9 (PIF1). *Starting from any configuration, Algorithm* PIF *satisfies* [PIF1] *of Specification 1.*

The following theorem is a corollary of Lemmas 8 and 9:

Theorem 2 (Snap PIF). *Algorithm* PIF *is snap-stabilizing.*

4 Complexity

Variable $S[0 \dots \delta - 1]$ which is an array of values in $\{0, \dots, M\}$ has the highest complexity: $\theta(\delta \times log(M))$ or $\theta(\delta \times log(k))$ since M is proportionnal to k. So the memory space needed on each processor is in $\theta(\delta \times log(k))$.

In order not to mix up the network's performances with the algorithm's performances, we assume in the rest of the paper that the only losses of messages are due to full channels, this case can appear when timers are faster than a round. However, as we will see, the round complexity is sensitive to the speed of the timers. So we will consider two cases, the first one is *slow timers* (with a period of the order of k rounds or a speed of the order of the speed of k rounds) and the second one is *fast timers* (with a period at most of the order of a round or a speed at least of the order of the round speed). We show that the round complexity of Algorithm *PIF* is in $\theta(h \times k)$ and $\theta((h \times k) + k^2)$ with slow and fast timers, respectively, Of course in order to get the best performance of a system the users have to adjust the speed of the timers to the expected average frequency of message losses.

Lemma 10 (Round Complexity of Message Delivering). *Let p and q be two neighboring processors. Let M be a message sent by p to q, then q will receive M in $\theta(k)$ rounds where k is the maximal capacity of the channels. More precisely, the number of rounds is less than or equal to $k + 1$.*

Proof. In the worst case Channel (p, q) is full and only one message is consumed from (p, q) by round. So M reaches q in $k + 1$ rounds. □

Corollary 2 (Round Complexity of Neighboring Broadcast). *Let p and q be two neighboring processors. Assume that p starts to send* ("B",0) *to q and any message* ("B",X) *($X \in \{0, \ldots, M-2\}$) from p is a working message for q, then p will receive the acknowledgement of its message* ("B",M-2) *in $\theta(k)$ rounds and $\theta(k^2)$ rounds with slow and fast timers, respectively, where k is the maximal capacity of the channels.*

Proof. In the worst case, when p starts to send *("B",0)*, the channel is full in both directions and contains neither *("B",0)* (in (p,q)) nor *("ACK",0)* (in (q,p)). Moreover each *("B",0)* is a working message for q. From Lemma 10, p will receive *("ACK",0)* in $2k+2$ rounds.

With slow timers when p receives the first *("ACK",0)*, (p,q) and (q,p) contain at most a constant number of *("B",0)* and *("ACK",0)*, respectively. So when p starts to send *("B",1)*, it will receive the first *("ACK",1)* in a constant number of rounds. By induction on X it is clear that p sends $\theta(1)$ times each of its $M-2$ messages *("B",X)* ($X \in \{1, \ldots, M-2\}$). Finally it will receive the acknowledgement of its message *("B",M-2)* after $\theta(k)$ rounds.

With fast timers when p receives the first *("ACK",0)*, (p,q) is full of *("B",0)* and (q,p) is full of *("ACK",0)*. So when p starts to send *("B",1)*, the situation is similar to that above and p will receive *("ACK",1)* in $\theta(k)$ rounds. By induction on X it is clear that p sends $\theta(k)$ times each of its $M-1$ messages *("B",X)* ($X \in \{0, \ldots, M-2\}$). Since $M = 2k+3$, it will receive the acknowledgement of its message *("B",M-2)* after $\theta(k^2)$ rounds. □

The following lemma can be formally proved for *bottom abnormal roots* by induction on the distance to the farthest leaf and the result can be extended to *top abnormal roots*.

Lemma 11 (Round Complexity of the Disappearance of the Abnormal Roots). *Starting from any configuration, the system will never contain* abnormal roots *in $\theta(h \times k)$ rounds, where h and k are the height of the tree and the maximal capacity of the channels, respectively.*

Let us call *parasite messages* the messages which do not concern the broadcast coming from r. After the disappearance of the abnormal roots, the only parasite messages can be *("B",M-1)*, *("ACK",X)*, *("F")*, *("F_ACK")*, *("B:No")*, *("Broadcast?")*, or *("Withdraw")*. Among those messages only *("B",M-1)*, *("F")*, and *("Broadcast?")* can generate some response: *("Withdraw")*, *("F_ACK")*, and *("B:No")*, respectively. So from Lemma 10 in $\theta(k)$ rounds, the system contains no parasite messages until a new broadcast starts from r and the channels between two processors satisfying *End* are empty whichever timer you use.

Now it is easy to see that the round complexity can be reduced to the round complexity of the propagation of the broadcast, the extra term for fast timers is due to the initial broacast in a full channel.

Theorem 3 (Round Complexity of Algorithm *PIF*). *Starting from any configuration, the round complexity of Algorithm* PIF *is in $\theta(h \times k)$ and $\theta((h \times$*

$k) + k^2)$ *rounds with slow and fast timers, respectively, where* h *is the height of the tree and* k *is the maximal capacity of the channels.*

We think that snap-stabilization of channels is unavoidable in order to get a snap-stabilizing algorithm in the message passing model, so we conjecture that Algorithm *PIF* is an optimal snap-stabilizing algorithm in terms of round complexity.

5 Conclusion

There exist a lot of approaches of the concept of self-stabilization, but to our knowledge, snap-stabilization is the only variant of self-stabilization which has been proved power equivalent to self-stabilization in the context of the state model (a locally shared memory model) and for non anonymous systems [11]. Moreover snap-stabilization enhances the safety of the system since the stabilization time is nul. So the problem of the existence of snap-stabilizing solutions in the message passing model is a very crucial question from a practical point of view. In this paper, we have presented the first snap-stabilizing propagation of information with feedback (PIF) protocol for non-oriented trees in the message passing model. Following the impossibility result in [16] we consider that the capacity of the channels is bounded. Using slow and fast timers, we show that the round complexity of our algorithm is in $\theta(h \times k)$ and $\theta((h \times k) + k^2)$, respectively, where h is the height of the tree and k is the maximal capacity of the channels. We conjecture that our algorithm is optimal in terms of round complexity. The next problem to solve is the design of a snap-stabilizing PIF algorithm for arbitrary networks and then to get the power equivalence with self-stabilization in the message passing model.

References

1. Y Afek and GM Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7(1):27–34, 1993.
2. Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction (extended abstract). In *32nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 268–277. IEEE Computer Society, 1991.
3. Lélia Blin, Alain Cournier, and Vincent Villain. An improved snap-stabilizing pif algorithm. In *Self-Stabilizing Systems, 6th International Symposium, SSS 2003, San Francisco, CA, USA, June 24-25, 2003, Proceedings*, Lecture Notes in Computer Science, pages 199–214. Springer, 2003.
4. A. Bui, A.K. Datta, F. Petit, and V. Villain. State-optimal snap-stabilizing pif in tree networks. In *Workshop on Self-stabilizing Systems (WSS)*, pages 78–85. IEEE Computer Society, 1999.
5. Alain Bui, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. Snap-stabilizing pif algorithm in the tree networks without sense of direction. In *SIROCCO'99, 6th International Colloquium on Structural Information & Communication Complexity, Lacanau-Ocean, France, 1-3 July, 1999*, pages 32–46. Carleton Scientific, 1999.

6. Alain Bui, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. Snap-stabilization and pif in tree networks. *Distributed Computing*, 20(1):3–19, 2007.
7. Ernest J. H. Chang. Echo algorithms: Depth parallel operations on general graphs. *IEEE Trans. Software Eng.*, 8(4):391–401, 1982.
8. A. Cournier, A.K. Datta, F. Petit, and V. Villain. Snap-stabilizing pif algorithm in arbitrary networks. In *22nd International Conference on Distributed Computing Systems (ICDCS)*, pages 199–208, 2002.
9. A. Cournier, S. Devismes, and V. Villain. Snap-stabilizing pif and useless computations. In *12th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 39–48. IEEE Computer Society, 2006.
10. Alain Cournier, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. Optimal snap-stabilizing pif in un-oriented trees. In *5th International Conference on Principles of Distributed Systems. OPODIS 2001, Manzanillo, Mexico, December 10-12, 2001. Proceedings*, Studia Informatica Universalis, pages 71–90, 2001.
11. Alain Cournier, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. Enabling snap-stabilization. In *23rd International Conference on Distributed Computing Systems (ICDCS 2003), 19-22 May 2003, Providence, RI, USA*, pages 12–19. IEEE Computer Society, 2003.
12. Alain Cournier, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. Optimal snap-stabilizing pif algorithms in un-oriented trees. *J. High Speed Networks*, 14(2):185–200, 2005.
13. Alain Cournier, Stéphane Devismes, and Vincent Villain. From self- to snap-stabilization. In *Stabilization, Safety, and Security of Distributed Systems, 8th International Symposium, SSS 2006, Dallas, TX, USA, November 17-19, 2006, Proceedings*, Lecture Notes in Computer Science, pages 199–213. Springer, 2006.
14. Alain Cournier, Stéphane Devismes, and Vincent Villain. Snap-stabilizing pif and useless computations. In *12th International Conference on Parallel and Distributed Systems (ICPADS 2006), 12-15 July 2006, Minneapolis, Minnesota, USA*, pages 39–48. IEEE Computer Society, 2006.
15. Alain Cournier, Stéphane Devismes, and Vincent Villain. Light enabling snap-stabilization of fundamental protocols. *TAAS*, 4(1):1–27, 2009.
16. Sylvie Delaët, Stéphane Devismes, Mikhail Nesterenko, and Sébastien Tixeuil. Snap-stabilization in message-passing systems. In *Distributed Computing and Networking, 10th International Conference, ICDCN 2009, Hyderabad, India, January 3-6, 2009. Proceedings*, Lecture Notes in Computer Science, pages 281–286. Springer, 2009.
17. Stéphane Devismes, Franck Petit, and Vincent Villain. Autour de l'autostabilisation 1. techniques généralisant l'approche. *Technique et Science Informatiques*, 30(7):873–894, 2011.
18. Stéphane Devismes, Franck Petit, and Vincent Villain. Autour de l'autostabilisation 2. techniques spécialisant l'approche. *Technique et Science Informatiques*, 30(7):895–922, 2011.
19. E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.
20. Shlomi Dolev and Nir Tzachar. Empire of colonies: Self-stabilizing and self-organizing distributed algorithms. In *Principles of Distributed Systems, 10th International Conference, OPODIS 2006, Bordeaux, France, December 12-15, 2006, Proceedings*, Lecture Notes in Computer Science, pages 230–243. Springer, 2006.
21. Adrian Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, 29(1):23–34, 1983.