State-Optimal Snap-Stabilizing PIF In Tree Networks

(Extended Abstract)

Alain Bui, [1] Ajoy K. Datta,[2] * Franck Petit,[1]
Vincent Villain[1]
[1] LaRIA, Université de Picardie Jules Verne, France
[2] Department of Computer Science, University of Nevada, Las Vegas

Abstract

In this paper, we introduce the notion of snap-stabilization. A snap-stabilizing algorithm protocol guarantees that, starting from an arbitrary system configuration, the protocol always behaves according to its specification. So, a snap-stabilizing protocol is a self-stabilizing protocol which stabilizes in 0 *steps.*

We propose a snap-stabilizing Propagation of Information with Feedback (PIF) scheme on a rooted tree network. We call this scheme Propagation of information with Feedback and Cleaning *(*$\mathcal{PFC}$*). We present two algorithms. The first one is a basic* $\mathcal{PFC}$ *scheme which is inherently snap-stabilizing. However, it can be delayed* $O(h^2)$ *steps (where* h *is the height of the tree) due to some undesirable local states. The second algorithm improves the worst delay of the basic* $\mathcal{PFC}$ *algorithm from* $O(h^2)$ *to* 1 *step. The* $\mathcal{PFC}$ *scheme can be used to implement the distributed reset, the distributed infimum computation, and the global synchronizer in* $O(1)$ *waves (or PIF cycles). Moreover, assuming that a (local) checking mechanism exists to detect transient failures or topological changes, the* $\mathcal{PFC}$ *scheme allows processors to (locally) "detect" if the system is stabilized, in* $O(1)$ *waves without using any global metric (such as the diameter or size of the network).*

Finally, we show that the state requirement for both $\mathcal{PFC}$ *algorithms matches the exact lower bound of the PIF algorithms on tree networks—*3 *states per processor, except for the root and leaf processors which use only* 2 *states. Thus, the proposed algorithms are optimal PIF schemes in terms of the number of states.*

Keywords: *Fault-tolerance, optimality, PIF, self-stabilization, snap-stabilization, synchronization.*

*Supported in part by a sabbatical leave grant from the University of Nevada, Las Vegas.

1. Introduction

The concept of self-stabilization [11] 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.

Chang [9] and Segall [16] defined the concept of *Propagation of Information with Feedback* (PIF) (also called *wave propagation*). The PIF scheme can be informally described as follows: A node initiates a wave, called the propagation wave. Every node, upon receiving this wave, forwards the wave to its neighbors, except the one it received the wave from. When a node receives an acknowledgment from all the neighbors it sent the propagation wave, it sends an acknowledgment, called the feedback wave, to the neighbor which originally sent it the propagation wave. So, eventually, the feedback wave reaches the processor which initiated the wave.

Related Work. Many distributed algorithms, e.g., distributed infimum function computations, synchronizers, termination detection, are based on a wave propagation scheme (see [17] for these algorithms). Self-stabilizing wave propagation protocols have been extensively used in the area of synchronizers [2, 4, 8, 18]. The counter flushing technique [18] provides a PIF scheme on a tree network. This scheme takes $O(1)$ waves to stabilize and takes $(\log n)$ extra space to implement the counters, where h is the height of the tree. A global self-stabilizing synchronizer for tree networks is proposed in [2]. The space complexity of the algorithm is 4 states (2 bits) and the time complexity is $O(h)$ waves. It is important to mention the work of Kruijer [13]. The space complexity of this algorithm is also 4 states. Although this paper did not discuss the problem of PIF, this algorithm can also be considered as a solution to the PIF in tree networks.

The solution proposed in [8] deals with the topology

changes, i.e., the dynamic systems. Protocols tolerating topological changes, e.g., [1, 6, 12], assume a local mechanism allowing processors to detect that a local topological change occurred. The ability to locally detect that something "wrong" happened (topology change, transient fault, etc.) has also been assumed in [3, 5, 7]. The most general method to "repair" the system is to reset the entire system after a transient fault is detected. Reset protocols are also wave-based algorithms.

Contributions. In this paper, we introduce the notion of snap-stabilization. A *snap-stabilizing protocol* guarantees that the system always maintains the desirable behavior. In other words, a snap-stabilizing algorithm is also a self-stabilizing algorithm which stabilizes in 0 steps. Obviously, any snap-stabilizing protocol is optimal in terms of the worst-case stabilization time.

We present a new PIF paradigm for the rooted tree networks. We call it the *Propagation of information with Feedback and Cleaning* ($\mathcal{PFC}$) scheme. It uses a key concept, called *cleaning*, introduced by Villain [19, 20]. We propose an algorithm which implements the basic $\mathcal{PFC}$ scheme, called Algorithm $\mathcal{PFC}$, without considering the property of snap-stabilization. Then we show that Algorithm $\mathcal{PFC}$ is already snap-stabilizing. However, the PIF cycle may start (in the worst case) after $O(h^2)$ steps, where h is the height of the tree. We improve this undesirable delay by presenting another algorithm, called *fast*$\mathcal{PFC}$ which reduces the delay to at most 1 step only.

One of the main advantages of the PIF scheme to be snap-stabilizing is the following: Assume that a (local) mechanism (as in [1, 3, 5, 6, 7, 12]) exists which allows processors to (locally) detect that a transient fault or a topological change occurred. With such an assumption, when the root initiates a round (a PIF cycle) during which no fault or topological change occurred, the root "detects" at the end of the round that the system is stabilized. So, using only the PIF waves, both algorithms presented in this paper allow the processors to locally decide whether the system is stabilized or not in $O(1)$ waves with no knowledge of any global metric (e.g., the network size or the height). The above scheme of local detection is applicable to problems like the distributed reset, the distributed infimum function computation, and the global synchronizer.

Another parameter to evaluate the efficiency of self-stabilizing algorithms is the *number of states* that each processor is required to have. The space requirement for both algorithms proposed in this paper is only 3 states per processor (only 2 states for the root and leaves). We show that this state requirement is the minimal state requirement for a PIF algorithm on a rooted tree.

Thus, both $\mathcal{PFC}$ algorithms stabilize in 0 step, and are state optimal implementation of the PIF scheme.

Outline of the Paper. In Section 2, we describe the distributed systems and the model we consider in this paper. In Section 3, we define the problem to be solved in this paper. The two $\mathcal{PFC}$ algorithms are presented in Section 4. The proof of the space optimality (in terms of the number of states per processor) is given in Section 5. Finally, we make some concluding remarks in Section 6.

2. Preliminaries

In this section, we define the distributed systems and programs considered in this paper, and state what it means for a protocol to be snap-stabilizing.

System. A *distributed system* is an undirected connected graph, $S = (V, E)$, where V is a set of nodes ($|V| = n$) and E is the set of edges. Nodes represent *processors* and edges represent *bidirectional communication links*. A communication link (p, q) exists iff p and q are neighbors. We consider networks which are *asynchronous* and *tree structured*. No processor, except the one, called the *root* and denoted by r, has any identity. We denote the set of *leaf* processors by L and the set of *internal* processors by I. We denote the set of processors in the tree, rooted at processor p, as T_p (hereafter, called the *tree* T_p). Note that $V = \{r\} \cup I \cup L = T_r$. $h(T_p)$ denotes the *height* of the tree rooted at p.

Each processor p maintains its set of neighbors, denoted as N_p. The *degree* of p is the number of neighbors of p, i.e., equal to $|N_p|$. We assume that each processor p ($p \neq r$) knows its *ancestor*, denoted by A_p. We assume that N_p and A_p are constants. In the remainder, we denote the set of *descendants* of any processor $p \in I \cup \{r\}$ by D_p, i.e., $D_p = N_p$ if $p = r$ and $D_p = N_p \setminus \{A_p\}$ if $p \in I$.

Programs. Every processor (except the root r) with the same degree executes the same program. This type of programs is known as a *semi-uniform distributed algorithm* [12]. The program 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 can only read its own variables and variables owned by the neighboring processors. So, the variables of p can be accessed by p and its neighbors.

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

The *state* of a processor is defined by the values of its variables. The *state* of a system is a product of the states of all processors ($\in V$). In the sequel, we refer to the state of a processor and system as a (*local*) *state* and *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 computations of a protocol $\mathcal{P}$ in system S starting with a particular configuration $\alpha \in \mathcal{C}$ is denoted by $\mathcal{E}_\alpha$. The set of all possible computations of $\mathcal{P}$ in system S is denoted as $\mathcal{E}$.

We assume a *distributed daemon*, i.e., during a computation step, one or more processors execute a step and a processor may take at most one step. A processor p is said to be *enabled* at γ $(\gamma \in \mathcal{C})$, if there exists an action A such that the guard of A is true in p at γ. We assume a *weakly fair* daemon, meaning that if a processor p is continuously $enabled$, p will be eventually chosen by the daemon to execute an action.

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

Definition 2.1 (Snap-stabilization) *The protocol $\mathcal{P}$ is snap-stabilizing for the specification $\mathcal{SP}_\mathcal{P}$ on $\mathcal{E}$ if and only if the following condition holds: $\forall \alpha \in \mathcal{C} \ : \forall e \in \mathcal{E}_\alpha \ :: e \vdash \mathcal{SP}_\mathcal{P}$.*

3. Specification of the Propagation of Information With Feedback

Let us quickly review the well-known *PIF scheme* [9, 16] on tree structured networks. The PIF scheme is the repetition of a *PIF cycle*. The PIF cycle can be informally defined as follows: Starting from an initial configuration where no message has been yet broadcast, the root (r) initiates the *broadcast* phase and its descendants (except the leaf processors) participate in this phase by forwarding the broadcast message to their descendants. Once the broadcast phase reaches the leaf processors, since the leaf processors have no descendants, they notify to their ancestor of the termination of the broadcast phase by initiating the *feedback* phase. Once every processor, except the root, sent the feedback message to its ancestor, the root executes a special internal action indicating the *termination* or completion of the current PIF cycle.

Based on the above description, we define the PIF cycle in terms of *B-actions*, *F-actions*, and a *T-action*. *B-actions* and *F-actions* refer to the actions executed during the *broadcast* and *feedback* phase, respectively. The *T-action* is a special internal action executed by the root to *terminate* the current PIF cycle. We will use the term *a PIF-action* to refer to a *B-action*, an *F-action*, or a *T-action*.

Specification 3.1 (PIF Cycle) *A finite computation $e \in \mathcal{E}$ is called a PIF Cycle, denoted by $e \vdash PIF\text{-}cycle$, if and only if the following conditions are true:*

[L1] *At least one processor p, called an initiator, sends a message or terminates (or completes) the PIF cycle ($PIF\text{-}action$).*

[L2] *If an internal processor p ($p \in I$) receives a broadcast message from its ancestor A_p, then p eventually sends a broadcast message to all its descendants ($B\text{-}action$).*

[L3] *If a leaf processor p ($p \in L$) receives a broadcast message from its ancestor A_p, then p eventually sends a feedback message to A_p ($F\text{-}action$).*

[L4] *If an internal processor p ($p \in I$) receives a feedback message from at least one of its descendants, then p eventually sends a feedback message to its ancestor A_p ($F\text{-}action$).*

[L5] *If the root r receives a feedback message from at least one of its descendants, then r eventually terminates the current PIF cycle ($T\text{-}action$).*

[S] *The root r cannot terminate the PIF cycle ($T\text{-}action$) more than once.*

Conditions [L1] to [L5] are called the *liveness* properties, and Condition [S] is called the *safety* property. Specification 3.1 is similar to the one proposed in [17].

From Condition [L1], there exists at least one initiator in a PIF cycle. Let $Init$ be the set of processors that initiate the PIF cycle. We will call a PIF cycle as a *normal PIF cycle* if $Init = \{r\}$ and the first action of r is a *B-action*. The following property follows from Specification 3.1:

Property 3.1 *In a normal PIF cycle (i.e., $Init = \{r\}$), every processor p executes a $T\text{-}action$, an $F\text{-}action$, and a $B\text{-}action$ at most once.*

From Property 3.1, it is obvious that once a processor p executes one of the three $PIF\text{-}action$s, p will not execute that $PIF\text{-}action$ any more in the same PIF cycle. We say that p is $B\text{-}done$, $F\text{-}done$, or $T\text{-}done$ to indicate that p has executed its corresponding action (B, F, or T, respectively) during the current PIF cycle. The following property follows from Specification 3.1 and Property 3.1:

Property 3.2 *In any configuration of the PIF cycle, one of the following two conditions is true: (i) The root r will be*

eventually enabled to execute a PIF-$action$, and (ii) r is surrounded by a border of processors, which will be eventually enabled to execute a PIF-action, such that every processor p between r and the enabled processors (including r) is B-$done$.

Property 3.2 implies that, starting from any configuration of a normal PIF cycle, the system may contain several initiators (see Condition [L1] of Specification 3.1) enabled to execute either a *B-action* or an *F-action*. The *B-actions* extend the broadcast phase until the information (message from the initiators) reaches a leaf processor initiating the feedback phase. Similarly, the *F-actions* extend the feedback phase until the root terminates the end of the PIF-cycle. Thus, any suffix of a PIF-cycle is also a PIF-cycle. We state this property formally in the following property:

Property 3.3 $\forall e \vdash PIF\text{-}cycle,\ \forall e', e'' : e = e'e'' :: e'' \vdash PIF\text{-}cycle$.

Specification 3.2 (PIF Scheme) *We define a computation e starting from a configuration α ($e \in \mathcal{E}_\alpha$) as a PIF scheme, denoted by $e \vdash \mathcal{SP}_{PIF}$, if e is an infinite sequence of PIF cycles $e_0, e_1, \ldots$ ($\forall i \geq 0 : e_i \vdash PIF\text{-}cycle$) such that $\forall i \geq 1$, e_i is a normal PIF cycle (i.e., $Init = \{r\}$).*

Specification 3.2 implies that the only possible PIF-$action$ following a T-$action$ is a B-$action$ by the root r. We assume that r executes the T-$action$ and B-$action$ in the same computation step, and we refer to this as an TB-$action$. Property 3.2 for the PIF cycle can now be extended to the PIF scheme as follows:

Property 3.4 *In any configuration of the PIF scheme, Property 3.2 is true.*

4. Propagation of Information With Feedback and Cleaning ($\mathcal{PFC}$)

4.1. The Basic Algorithm

Algorithm $\mathcal{PFC}$, an implementation of the PIF scheme (as defined in Specification 3.2) is shown in Figure 4.1. Every processor p maintains a variable S_p, called the *state variable* of p. An internal processor ($\in I$) can have three different state values C, B, or F. The value C denotes the *initial* state of any processor before it participates in the PIF-cycle. The idea of this new state, called the *cleaning* state was introduced by Villain [19]. The state value B or F means that the processor has executed its B-$action$ or F-$action$, respectively. An internal processor executes its B-$action$ (respectively, F-$action$) by changing its state variable from C to B (respectively, from B to F). The leaf processors use only F and C, and execute their F-$action$ by changing their state from C to F. The root also uses only two state values: B and C. The root executes its TB-$action$ and C-$action$ by changing its state from C to B and B to C, respectively.

Algorithm 4.1 ($\mathcal{PFC}$) PIF in Rooted Tree Networks.

Variable

S_i ($i \in I$) $\in \{B, F, C\}$ for the internal processors.
$S_r \in \{B, C\}$ for the root.
S_l ($l \in L$) $\in \{F, C\}$ for the leaf processors.

Actions

{**For the internal processors**}

IB-$action$ :: $S_p = C \wedge S_{A_p} = B \wedge (\forall d \in D_p :: S_d = C) \longrightarrow S_p := B$;

IF-$action$ :: $S_p = B \wedge (\forall d \in D_p :: S_d = F) \longrightarrow S_p := F$;

IC-$action$ :: $S_p = F \wedge (\forall q \in N_p :: S_q \in \{F, C\}) \longrightarrow S_p := C$;

{**For the root**}

rTB-$action$ :: $S_p = C \wedge (\forall q \in N_p, S_q = C) \longrightarrow S_p := B$;

rC-$action$ :: $S_p = B \wedge (\forall q \in N_p, S_q = F) \longrightarrow S_p := C$;

{**For the leaf processors**}

LF-$action$ :: $S_p = C \wedge S_{A_p} = B \longrightarrow S_p := F$

LC-$action$ :: $S_p = F \wedge S_{A_p} \in \{F, C\} \longrightarrow S_p := C$;

According to the PIF cycle specification, the normal *broadcast* phase is followed by the *feedback* phase which is initiated by the leaf processors ([L3] and LF-$action$). After initiating the *feedback* phase, in the next step, the leaf processors can initiate the *cleaning* phase by changing its state from F to C (LC-$action$). So, the feedback and the cleaning phase can run concurrently. All we have to make sure is that the *cleaning* phase does not meet the broadcast phase, i.e., the processors in the *cleaning* phase do not confuse the processors in the *broadcast* phase. We implement this constraint as follows: An internal processor can execute its C-$action$ (i.e., changes its state from F to C) only if all its neighbors are in the *feedback* or *cleaning* phase (i.e., in state F or C). Thus, as soon as the ancestor of a leaf processor executes its F-$action$ (i.e., changes its state from B to F), the leaf processor can execute its C-$action$.

Eventually, the *feedback* phase reaches the descendants of the root. The root now executes its rC-$action$, i.e., changes its state from B to C. The root then waits until all its descendants are in the *cleaning* phase. Next, the root changes its state from C to B (TB-$action$). This marks the end of the current PIF cycle and the start of the next cycle.

Since our solution requires three phases (*broadcast*, *feedback*, and *cleaning*) in the PIF cycle, we call our method the *Propagation of Information with Feedback and Cleaning* ($\mathcal{PFC}$) and the corresponding cycle the *$\mathcal{PFC}$ cycle*.

Due to asynchrony in the network, some processors may be involved in a $\mathcal{PFC}$ cycle whereas the others are still executing the *cleaning* phase of the previous $\mathcal{PFC}$-cycle. But, we need to make sure that these two cycles do not confuse

each other. We solve this problem by adding two preventives in the algorithm. (i) A processor is allowed to execute its C-$action$ only if all its neighbors are in the *feedback* or *cleaning* phase. As we explained above, it prevents the *cleaning* phase to meet the *broadcast* phase. (ii) A processor can execute the B-$action$ only after all its descendants execute their C-$action$ (while its ancestor is in the *broadcast* phase). Thus, the processors which are slow to execute their C-$action$, are protected from the *broadcast* phase of the next $\mathcal{PFC}$ cycle.

4.2. Algorithm $\mathcal{PFC}$: A Snap-Stabilizing PIF Scheme

We claim that Algorithm $\mathcal{PFC}$ is a snap-stabilizing PIF scheme, i.e., starting from any configuration, it satisfies the specification of the PIF scheme (Specification 3.2).

Informally, the system is in an "*abnormal*" configuration if there exists at least an internal processor p ($\in I$) such that $S_p = B$ and $S_{A_p} \neq B$. This local configuration is called an *abnormal local configuration*. In such a configuration, A_p in not enabled. While $S_p = B$, it is easy to observe (by Algorithm $\mathcal{PFC}$ and by induction on every abnormal local configuration in T_p) that (i) the $\mathcal{PFC}$ cycle in T_p eventually completes, and (ii) once the $\mathcal{PFC}$ cycle in T_p completes, no abnormal local configuration exists in the system. So, eventually, the condition $\forall q \in D_p, S_q = F$ becomes true. Then, p will execute the IF-$action$ and the abnormal local configuration will disappear. Hence, in the worst case, the abnormal local configuration in p disappears after a complete $\mathcal{PFC}$ cycle. This process will be repeated until all abnormal local configurations disappear.

Now, consider a processor p such that the value of $h(T_p)$ is one of the maximum among all processors which are in an abnormal configuration. Then, every processor q on the path from the root r to p has $S_q = B$. Thus, irrespective of the initial configuration, every computation satisfies Property 3.4. The abnormal configurations can just delay the initiation of the first PIF cycle. This leads to the following theorem:

Theorem 4.1 *Algorithm $\mathcal{PFC}$ is snap-stabilizing.*

It is obvious from the above discussion that the delay to start the first $\mathcal{PFC}$ cycle depends on how quickly all the abnormal local configurations can be removed. The removal of an abnormal local configuration on p depends on the duration of the $\mathcal{PFC}$ cycle rooted at T_p. The worst case is the existence of a path from the root to a leaf node, which has the following sequence of states: $(BC)^{\alpha}C$ or $(BC)^{\alpha}$ ($\alpha > 1$), depending on the length of the path being even or odd. In this case, every processor p in state C has to wait for all its descendants to become F, before p can extend the broadcast phase from its ancestor to its descendants. Obviously, in the worst case, the delay is $O(h^2)$ steps. Thus, we can claim the following result:

Theorem 4.2 *The delay to start the first PIF cycle using Algorithm $\mathcal{PFC}$ is bounded by $O(h^2)$ steps.*

We will now improve Algorithm $\mathcal{PFC}$ to reduce the above delay to only 1 step.

4.3. Fast Snap-Stabilizing Algorithm $\mathcal{PFC}$

We make two modifications (shown below) in Algorithm $\mathcal{PFC}$ to obtain the fast snap-stabilizing PIF algorithm, called *fast*$\mathcal{PFC}$. First, we add a *correction action* called $ICorrection$. When an internal processor p finds that the relation between its state and that of its ancestor is not correct, i.e., the system is in an abnormal configuration, then the internal processor executes Action $ICorrection$ to correct its state. Second, we modify Action IF-$action$ by adding the condition $(S_{A_p} = B)$ to make Actions IF-$action$ and $ICorrection$ mutually exclusive.

$$IF\text{-}action :: S_p = B \wedge S_{A_p} = B \wedge (\forall d \in D_p :: S_d = F) \longrightarrow S_p := F;$$

$$ICorrection :: \quad S_p = B \wedge S_{A_p} \in \{F, C\} \longrightarrow S_p := C;$$

Obviously, like Algorithm $\mathcal{PFC}$, Algorithm *fast*$\mathcal{PFC}$ also satisfies Property 3.4.

Theorem 4.3 *Algorithm* fast$\mathcal{PFC}$ *is snap-stabilizing.*

Now, let us consider the worst delay of the first $\mathcal{PFC}$ wave of Algorithm *fast*$\mathcal{PFC}$. Consider an abnormal local configuration on p, i.e., $S_{A_p} \in \{F, C\}$ and $S_p = B$. A_p is not enabled because at least one of its descendants (p) is in state B. Hence, A_p cannot execute any action while p does not change its state. The only way p can change its state in the current configuration is by executing $ICorrection$. Also, the guard of $ICorrection$ does not depend on the status of p's descendants. So, the time to remove the abnormal local configurations depends only on the execution of $ICorrection$ by every processor enabled to execute this action. It is easy to observe that the concurrent executions of $ICorrection$ by processors in different branches of the tree do not affect each other. Thus, we can assume that all the processors enabled to execute $ICorrection$, execute $ICorrection$ during the same computation step. Once p executes $ICorrection$, S_p becomes C and A_p may be enabled to execute an action. So, we can claim the following result:

Theorem 4.4 *The delay to start the first PIF cycle using Algorithm* fast$\mathcal{PFC}$ *is bounded by* 1 *step.*

5. State Optimality

In this section, we will show that both algorithms proposed in this paper are optimal PIF algorithms on a tree (as defined in Specification 3.2) in the number of states per processor. We will first consider a special type of tree networks which have only *one leaf processor* ($|L| = 1$), henceforth referred to as a (*linear*) *chain*. We can describe any 2-state algorithm on a chain using a set of *atomic actions*. The *atomic actions* are those that cannot be splitted into other actions. For every internal processor p, each atomic action is of the following form: $a(b)c \rightarrow d$, where $a, b, c, d \in \{0, 1\}$, a, b, and c are the state of the left neighbor of p, the current state of p, and the state of the right neighbor of p, respectively. d is the new value assigned to the state of p. Assume that the root and the leaf are the two extreme end processors, left and right, respectively. The atomic actions of the root and leaf are $(b)c \rightarrow d$ and $a(b) \rightarrow d$, respectively.

Snap-Stabilizing PIF Cycle on a 3-Processor Linear Chain with 2 States. Algorithm 5.1 (see [19] for details) is a general PIF scheme for 3-processor chains with 2 states per processor. a, b, c denote three constants, all of which $\in \{0, 1\}$. For every $x \in \{a, b, c\}$, $\bar{x}$ denotes $(x + 1)\ mod\ 2$. $< B >$ and $< F >$ refer to the *B-actions* and *F-actions* of the processors, respectively.

Algorithm 5.1 Two State Snap-Stabilizing PIF Scheme for 3-Processor Chains.

$\forall (a, b, c) \in \{0, 1\}^3 ::$
{**For the root**p_1}
$R_1 :: (a)b \rightarrow \bar{a}\ < B >$ $\quad R_2 :: (\bar{a})\bar{b} \rightarrow a$
{**For the internal processor**p_2}
$N_1 :: \bar{a}(b)c \rightarrow \bar{b}\ < B >$ $\quad N_2 :: a(\bar{b})\bar{c} \rightarrow b\ < F >$
{**For the leaf processor**p_3}
$L_1 :: \bar{b}(c) \rightarrow \bar{c}\ < F >$ $\quad L_2 :: b(\bar{c}) \rightarrow c$

Note that, since the system has only three processors, any configuration of the system can be represented as $s_1 s_2 s_3$, where $\forall i \in [1, 3]$, s_i denotes the state of processor p_i ($\mathcal{C} = \{0, 1\}^3$). Figure 5.1 shows the complete behavior of Algorithm 5.1. We can easily observe the following result from Figure 5.1:

Theorem 5.1 $\forall (a, b, c) \in \{0, 1\}^3 : \forall e \in \mathcal{E}_{abc} :: e \vdash \mathcal{SP}_{PIF}$.

Note that every execution e, starting from any configuration $\alpha \in \mathcal{C}$ ($e \in \mathcal{E}_\alpha$), satisfies $\mathcal{SP}_{PIF}$. Hence, Algorithm 5.1 is trivially snap-stabilizing.

We will now show that, assuming a weakly fair central daemon, Algorithm 5.1 is the "unique" scheme to implement the PIF scheme on a 3-processor chain using only 2 states per processor. It is *unique* in the sense that any PIF algorithm written for such a chain is an instantiation of Algorithm 5.1 with a particular set of values for abc.

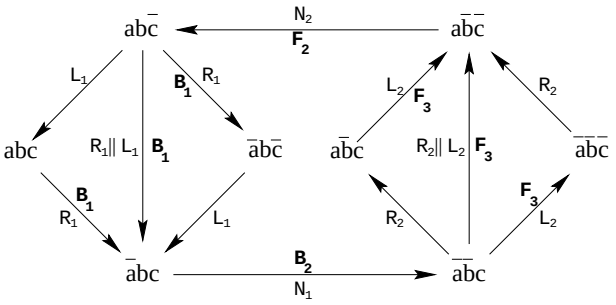

Figure 5.1. General 2-state PIF Scheme for a linear chain with three processors.

We use $\mathcal{A}_{(3p/2s)}$ to denote the set of 2-state algorithms on a 3-processor chain. Assuming a central daemon, any algorithm $A \in \mathcal{A}_{(3p/2s)}$ can be modeled as a subgraph τ_A of the transition graph T of all possible configurations, as shown in Figure 5.2.

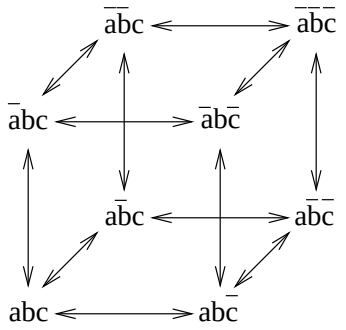

Figure 5.2. T: The Transition Graph of a 3-processor chain using 2 states.

Definition 5.1 *An elementary configuration path ρ is a finite sequence of configurations $\gamma_0, \gamma_1, \ldots, \gamma_t$ $(t > 0)$ such that (i) $\forall i \in [0, t[: \gamma_i \mapsto \gamma_{i+1}$ and (ii) $\forall i, j \in [0, t[: (\gamma_i = \gamma_j) \Leftrightarrow (i = j)$. The length of the elementary path ρ is denoted by $\sharp\rho$ and is equal to t.*

Definition 5.2 *An elementary configuration cycle σ is an elementary configuration path $\rho = \gamma_0, \gamma_1, \ldots, \gamma_t$ such that $\gamma_0 = \gamma_t$.*

As T is a hypercube of dimension 3, we can make the following remarks:

Remark 5.1 *The maximum length of an elementary configuration path ρ and an elementary configuration cycle σ in T is* 8.

Remark 5.2 *The length of any configuration cycle σ in T is even.*

We refer to an elementary configuration cycle σ as an x-cycle, where x is even and $x = \sharp\sigma$ (Remark 5.2). For example, in Figure 5.2, the cycles $\sigma_1 = abc, \bar{a}bc, abc$, $\sigma_2 = abc, ab\bar{c}, \bar{a}b\bar{c}, \bar{a}bc, abc$, and $\sigma_3 = abc, ab\bar{c}, \bar{a}b\bar{c}, \bar{a}\bar{b}\bar{c}, \bar{a}\bar{b}c, a\bar{b}c, abc$ are a 2-cycle, a 4-cycle, and a 6-cycle, respectively.

Remark 5.3 *In an x-cycle, $x = 2y$, $y \in [1, 3]$, only y processors execute an action.*

Let $\mathcal{PIF}_{(3p/2s)}$ be the subset of $\mathcal{A}_{(3p/2s)}$ such that every execution e of A, denoted by e_A, satisfies $\mathcal{SP}_{PIF}$ (as defined in Specification 3.2), i.e. $\forall A \in \mathcal{PIF}_{(3p/2s)} :: e_A \vdash \mathcal{SP}_{PIF}$. The following lemma follows from Remark 5.3 and Specification 3.1:

Lemma 5.1 *The length of every PIF cycle σ ($\sigma \vdash$ PIF cycle) is greater than or equal to* 6.

Lemma 5.2 *Consider a configuration $\alpha \in \tau_A$ $A \in \mathcal{PIF}_{(3p/2s)}$. If α is in a configuration cycle σ such that at least one PIF-$action$ belongs to σ, then σ is a PIF cycle.*

Proof. Assume that σ is not a PIF cycle. Since $\alpha \in \tau_A$, there may exist an execution e_A of Algorithm A such that $e_A = e'\sigma^{\omega}e''$, i.e., σ may be executed at least twice successively. Thus, e_A does not satisfy $\mathcal{SP}_{PIF}$ which contradicts $A \in \mathcal{PIF}_{(3p/2s)}$. □

Lemma 5.3 *Every PIF cycle σ of τ_A, $A \in \mathcal{PIF}_{(3p/2s)}$ is a 6-cycle.*

Proof. Assume that the computation step $abc \mapsto \bar{a}bc$ belongs to τ_A. So, $ab\bar{c} \mapsto \bar{a}b\bar{c}$ belongs to τ_A because p_1 cannot distinguish the configurations abc and $ab\bar{c}$. Since $A \in \mathcal{PIF}_{(3p/2s)}$, there exists at least one action in p_1 to change a to $\bar{a}$ and another action to change $\bar{a}$ to a. (Otherwise, at least one computation does not satisfy $\mathcal{SP}_{PIF}$ which contradicts $A \in \mathcal{PIF}_{(3p/2s)}$.) Moreover, at least one of these two actions is the B-$action$ of p_1. So, by Lemmas 5.1 and 5.2, neither $\bar{a}bc \mapsto abc$ nor $\bar{a}b\bar{c} \mapsto ab\bar{c}$ belongs to τ_A. Thus, $abc \mapsto \bar{a}bc$, $ab\bar{c} \mapsto \bar{a}b\bar{c}$, $\bar{a}\bar{b}c \mapsto a\bar{b}c$, and $\bar{a}\bar{b}\bar{c} \mapsto a\bar{b}\bar{c}$ belong to τ_A.

We now consider two cases for p_3.

1. Assume that $abc \mapsto ab\bar{c}$ belongs to τ_A. Then, the four computation steps $abc \mapsto ab\bar{c}$, $\bar{a}bc \mapsto \bar{a}b\bar{c}$, $\bar{a}\bar{b}\bar{c} \mapsto \bar{a}\bar{b}c$, and $a\bar{b}\bar{c} \mapsto a\bar{b}c$ also belong to τ_A. Since $A \in \mathcal{PIF}_{(3p/2s)}$, there exists at least one action in p_2 to change b to $\bar{b}$ and another action to change $\bar{b}$ to b. By verifying every possible case of at least two actions to change b to $\bar{b}$ and $\bar{b}$ to b, we find that every configuration containing an elementary configuration cycle never contains a 8-cycle (i.e., it contains a 2-cycle, a 4-cycle, or a 6-cycle). By Lemma 5.1, the only possible PIF cycles are the 6-cycles.

2. Assume that $ab\bar{c} \mapsto abc$ belongs to τ_A. Following the same reasoning as above leads to the same conclusion.

The above result is true for any $abc \in \{0, 1\}^3$. □

The following theorem follows from Lemmas 5.1, 5.2, and 5.3.

Theorem 5.2 *Algorithm 5.1 is the unique scheme to implement a 2-state PIF scheme on a 3-processor chain under a central daemon.*

Any distributed daemon can behave like a central daemon by choosing only one enabled processor in each configuration. This leads to the following result:

Theorem 5.3 *Algorithm 5.1 is the unique scheme to implement a 2-state PIF scheme on a 3-processor chain under a distributed daemon.*

Minimal Number of Configurations on a Rooted Tree Network. We first prove that there exists no 2-state algorithm to implement the PIF scheme on an n-processor chain ($n > 3$). This result then leads to the fact that no such algorithm exists for the rooted trees with a height greater than 2 ($h(T_r) > 2$).

Lemma 5.4 *There exists no 2-state algorithm to implement the PIF scheme on an n-processor chain ($n > 3$).*

Proof Outline. Assume that such an algorithm exists. Since the algorithm works for n processors, it also works for 3 processors. So, for any three adjacent processors p_{i-1}, p_i, p_{i+1} ($i \in [1, n-1]$), the behavior of p_i is similar to that of p_2 of a 3-processor chain p_1, p_2, p_3. (Otherwise, we could write algorithms other than Algorithm 5.1 on a 3-processor chain). So, every internal processor (i.e., every processor except p_1 and p_n) executes Actions $N1$ and $N2$ of Algorithm 5.1. Thus, there are two cases: (i) $N1$ and $N2$ are asymmetric (case where $a \neq c$ in Algorithm 5.1), or (ii) $N1$ and $N2$ are symmetric (i.e., $a = c$). We can show, by checking all possible cases, that in both cases (i) and (ii), the system either reaches a deadlock situation (no action is enabled) or violates the specification $\mathcal{SP}_{PIF}$. □

Our final result now trivially follows from Lemma 5.4.

Corollary 5.1 *There exists no 2-state algorithm to implement the PIF scheme on a rooted tree T_r where $h(T_r) > 2$.*

Theorem 5.4 *Both Algorithm $\mathcal{PFC}$ and Algorithm* fast$\mathcal{PFC}$ *are optimal in terms of the number of states per processor.*

6. Conclusions

The $\mathcal{PFC}$ paradigm is a new approach to designing the PIF scheme. This scheme is optimal in space and in time. We proposed two algorithms to implement the $\mathcal{PFC}$ scheme. Both are snap-stabilizing, i.e., they are self-stabilizing and stabilize in 0 steps. The proposed algorithms can be used to implement the distributed reset, the distributed infimum function, and the global synchronizer in $O(1)$ waves (PIF cycle). Moreover, assuming the use of a (local) mechanism to detect the transient failures or topological changes, both algorithms allow processors to (locally) "detect" if the system is stabilized in $O(1)$ waves, without any global metric (e.g., the diameter or the size of the network).

The $\mathcal{PFC}$ scheme uses the cleaning phase strategy introduced by Villain [19, 20]. The cleaning phase strategy has also been used in [14, 15] to implement a depth-first token circulation on tree networks. The space optimality of [14] is proven in [15]. The problem of the minimal state requirement in general graphs is still open. The best known solution to this problem was presented in [10].

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 and R. Ostrovsky. Memory-efficient and self-stabilizing network reset. In *PODC94 Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 254–263, 1994.

[6] 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.

[7] B. Awerbuch, B. Patt-Shamir, G. Varghese, and S. Dolev. Self-stabilizing by local checking and global reset. In *WDAG94 Distributed Algorithms 8th International Workshop Proceedings, Springer-Verlag LNCS:857*, pages 326–339, 1994.

[8] 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.

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

[10] A. Datta, C. Johnen, F. Petit, and V. Villain. Self-stabilizing depth-first token circulation in arbitrary rooted networks. In *SIROCCO'98, The 5th International Colloquium On Structural Information and Communication Complexity Proceedings*, pages 229–243, 1998.

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

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

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

[14] F. Petit. Highly space-efficient self-stabilizing depth-first token circulation for trees. In *OPODIS'97, International Conference On Principles Of Distributed Systems Proceedings*, pages 221–235, 1997.

[15] F. Petit and V. Villain. Optimality and self-stabilisation over tree networks. *Parallel Processing Letters*, 1998. To appear.

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

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

[18] G. Varghese. Self-stabilization by counter flushing. In *PODC94 Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 244–253, 1994.

[19] V. Villain. A new lower bound for self-stabilizing mutual exclusion algorithms. Technical Report RR97-17, LaRIA, University of Picardie Jules Verne, 1997.

[20] V. Villain. New lower bounds of self-stabilizing mutual exclusion algorithms. Technical Report RR98-08, LaRIA, University of Picardie Jules Verne, 1998. also, presented at Dagstuhl Workshop on Self-Stabilization, August 17-21, 1998, Germany.