Available online at www.sciencedirect.com
ScienceDirect
J. Parallel Distrib. Comput. 67 (2007) 1 – 12
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

# Optimal snap-stabilizing depth-first token circulation in tree networks

Franck Petit*, Vincent Villain

*LaRIA, CNRS FRE 2733, Université de Picardie Jules Verne, Amiens, France*

Received 7 May 2005; received in revised form 12 July 2006; accepted 7 August 2006
Available online 30 October 2006

**Abstract**

We address the depth-first token circulation (DFTC) in tree networks. We first consider oriented trees—every processor knows which of its neighbors leads to a particular processor called the root. On such trees, we propose a state optimal DFTC algorithm. Next, we propose a second algorithm, also for trees, but where no processor knows which of its neighbor leads to the root. This algorithm is also optimal in terms of the number of states per processor.

Both algorithms works under any daemon, even unfair. Furthermore, both are *snap-stabilizing*. 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. Thus, both algorithms are also optimal in terms of the stabilization time.

Finally, two approaches of the maximum waiting time to initiate a DFTC are also discussed, whether the tree is oriented or not. In every case but one, we show that the waiting time is asymptotically optimal. In the last case, we conjecture the same result.

*Keywords:* Distributed mutual exclusion; Optimality; Self-stabilization; Snap-stabilization; Token passing

## 1. Introduction

The *depth-first token circulation problem* (DFTC) is to implement a token circulation scheme where the token is passed from one processor to another in the depth-first order such that every processor gets the token at least once in every token circulation cycle. This scheme has many applications in distributed systems, e.g., mutual exclusion, spanning tree construction, synchronization, and finding the biconnected components.

The concept of *snap-stabilization* was introduced in [3]. Snap-stabilization is related to *self-stabilization* [9]. Regardless of the initial states of the distributed system, a self-stabilizing algorithm is guaranteed to converge to the intended behavior in finite time, whereas a snap-stabilizing algorithm guarantees that it always behaves according to its specification. So, 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.

### 1.1. Related work

The first deterministic self-stabilizing DFTC algorithm for tree networks has been proposed by Dolev et al. [10]. The authors combined their self-stabilizing DFTC algorithm with an algorithm to construct a spanning-tree to design a self-stabilizing token circulation in an arbitrary rooted network. The token circulation algorithm in [10] requires the spanning-tree to be oriented, i.e., every processor knows which of its neighbors leads to a particular processor called the root. There exists many self-stabilizing spanning tree construction algorithms in the literature, e.g., [1,2,4,5].

The (depth-first) token circulation protocols can be used to compute a global task, to maintain a global structure, or a global property of a system (e.g., the mutual exclusion among the processors or the computation of a compact routing table). Any processor can initiate a global computation, e.g., infimum computation, naming, counting. Designing algorithms for un-oriented trees allows to maintain only one spanning tree (instead of one per processor for oriented trees) in any global computation in arbitrary networks, regardless of the number of the global computations and the corresponding initiators.

* Corresponding author.
*E-mail address:* petit@laria.u-picardie.fr (F. Petit).

0743-7315/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2006.08.008

We detail the interest of designing algorithms for un-oriented trees below.

To deal with the concurrent executions of the same global task, the identity of the associated network component (e.g., the resource or the root identity) must be added to the messages exchanged in the network, and also, must be maintained by every processor. So, each processor in the network has to maintain $n$ spanning trees rooted at $n$ distinct processors of the network ($n$ being the number of processors in the network). Thus, for each spanning tree, every processor needs to maintain a parent pointer, and a subset of children—the neighbors which are not the parent. There are $\Delta_p$ possible states for each parent pointer (or $\log \Delta_p$ bits), where $\Delta_p$ is the degree of the processor $p$. In the best case, each subset of children requires $2^{\Delta_p}$ states (or $\Delta_p$ bits, i.e., one bit per processor). Therefore, the total memory requirement of each processor $p$ is $\Omega((2^{\Delta_p})^n)$ states (or $\Omega(\Delta_p \times n)$ bits) in order to maintain $n$ spanning trees.

On the other hand, to design a DFTC for the above situation, i.e., to deal with the multiple initiators, if we use an un-oriented tree, every processor needs to maintain *only one spanning tree* of the network (instead of $n$ spanning trees). So, every processor $p$ needs a space of $2^{\Delta_p}$ states (or $\Delta_p$ bits) in order to maintain the spanning tree.

Note that in the worst case, $n$ tokens may be circulating simultaneously in the network. It follows that the total amount of memory on each processor $p$ is of the same order whether the tree needs to be oriented or not, i.e., $O(\Delta_p \times n)$ bits. However, in the latter case, the $O(\Delta_p)$ bits used to maintain the circulation of each token can be dynamically allocated. In contrast, the use of an oriented tree requires a static amount of memory depending of $n$.

A deterministic self-stabilizing DFTC algorithm for un-oriented tree networks has been first proposed in [16]. The DFTC of [16] uses a new token definition [12,20,21]. In these papers, the authors define the token as "the ability for a processor to make some *particular* moves", whereas Dijkstra originally defined the token as "the ability for a processor to make a move". So, the token definition of [12,20,21] relaxes Dijkstra's original token definition. This new notion of token allows processors to move concurrently while preserving the mutual exclusion property.

In [19], we show that at least $\Delta_p + 1$ states per processor $p$ are required to make the DFTC in oriented tree networks, where $\Delta_p$ is the degree of the processor $p$. Our result is valid even if the system tolerates no transient faults (in the remainder also referred to as "safe" system), i.e., the algorithm needs to be correctly initialized.

In the same paper [19], we show that at least $\Delta_p + 2$ states per processor $p$ are required to make the DFTC in un-oriented tree networks, provided that $p$ is not the root $\mathtt{R}$ and $\Delta_p \geqslant 3$. (If $p = \mathtt{R}$ or $\Delta_p < 3$, the minimal number of states per processor remains $\Delta_p + 1$ only, as in the oriented trees.) In [6], we show that, if $\Delta_p > 1$, 4 states are necessary to design a snap-stabilizing PIF (Propagation of Information with Feedback) algorithm in an un-oriented tree, in particular when $\Delta_p = 2$—only 2 states if $\Delta_p = 1$. When $\Delta_p = 2$, it is easy to see that $p$ behaves similarly by executing either the PIF or the DFTC scheme. So, this result is also valid to achieve the property of snap-stabilization for the DFTC problem. Thus, both results of [19] and [6] prove that if $\Delta_p > 1$, then $\Delta_p + 2$ states per processor $p$ are necessary to design any snap-stabilizing DFTC algorithm in un-oriented rooted tree networks—2 if $\Delta_p = 1$.

Most of works in the area of self-stabilization are proven assuming some type of *daemon*—or *scheduler*. The daemon is usually regarded as a *fair* adversary which activates either one (central daemon) or more distributed daemon processors among enabled processors. The daemon is fair means that if a processor is continuously enabled, then the daemon eventually activates the processor. Until now, only the DFTC algorithm for arbitrary networks in [7] works assuming an unfair daemon. All the other self-stabilizing DFTC in the current literature, either for tree [10,16] or arbitrary networks [13,15,18,14,8,17], assume a fair daemon. Furthermore, none of them is snap-stabilizing, except the one in [7]. However, it requires $n^n \times \Delta_p^2$ states per processor.

### 1.2. Contributions

In this paper, we propose two depth-first token circulation algorithms for tree networks. We prove that both algorithms are snap-stabilizing. The proof is made assuming an *unfair* distributed daemon, i.e., even if a processor $p$ is continuously enabled, then $p$ may never be chosen by the daemon, unless $p$ is the only enabled processor in the system. So, both works under any (fair or unfair, central or distributed) daemon. Moreover, both algorithms preserve the state optimality proven in [19] and [6], i.e., $\Delta_p + 1$ states per processor $p$, $\Delta_p + 2$ when $p$ is an internal processor of an un-oriented tree network.

Because snap-stabilizing algorithms stabilize in zero steps, the notion of (*maximum*) *waiting time* naturally comes up, i.e., the maximal time the root has to wait (starting from any configuration) to initiate a DFTC. This was never discussed previously in the area of self-stabilization. This comes from the fact that the waiting time was always merged into the stabilization time. Note that the waiting time can also be considered in a safe environment (without abnormal local state). It can be seen as the service time for the initiator (or root) of a DFTC, computed in terms of rounds instead of service delivering. Two approaches of the waiting time are discussed in Section 5, whether the tree is oriented or not. In any case but one, we show that the waiting time is asymptotically optimal. In the last case, we conjecture the same result.

### 1.3. Outline of the paper

In the next section (Section 2), we describe the distributed systems and the model we consider in this paper. We also present the notion of snap-stabilization there, followed by the specification of the DFTC. Section 3 addresses the DFTC algorithm for oriented trees. The DFTC algorithm for un-oriented trees is presented in Section 4. We discuss the waiting time for both algorithms in Section 5. Finally, we conclude this paper by 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.

### 2.1. System

A *distributed system* $S$ consists of $n$ processors. Each processor $p$ can communicate with some other processors, called neighbors of $p$. A (*communication*) *network* is an arbitrary undirected connected graph $G = (V, E)$ where the vertices in $V$ are the processors of $S$ and the edges in $E$ represent the bidirectional communication links of each pair of neighbors $(p, q)$. Every processor $p$ can distinguish all its links. To simplify the presentation, we refer to a link $(p, q)$ of processor $p$ simply by the *label* $q$. We assume that the labels, stored in the set $N_p$, are arranged in some arbitrary strict order $\succ_p$. We assume that $N_p$ is a constant and is maintained by an underlying protocol. The number of neighbors of $p$, $|N_p|$, is called the *degree* of $p$ and is denoted by $\Delta_p$.

### 2.2. Tree networks

We consider *asynchronous rooted tree* networks. We denote the root processor by $\mathtt{R}$. We denote the set of processors in the tree, rooted at processor $p$, as $T_p$ (hereafter, called the *tree* $T_p$). We denote the set of *leaf* processors by $L$ and the set of *internal* processors by $I$. Note that $V = \{\mathtt{R}\} \cup I \cup L = T_{\mathtt{R}}$. In the remainder, $h$ will denote the *height* of the tree rooted at $\mathtt{R}$, $T_{\mathtt{R}}$. Every processor $p$ which is not the root ($p \neq \mathtt{R}$) has a single *parent*, denoted by $P_p$, such that $P_p$ is the neighbor of $p$ ($P_p \in N_p$) on the path from $p$ to $\mathtt{R}$ if $\mathtt{R} \notin N_p$, or $P_p = \mathtt{R}$ otherwise. We denote the set of *children* of any processor $p \in I \cup \{\mathtt{R}\}$ by $C_p$, i.e., $C_p = N_p$ if $p = \mathtt{R}$, $C_p = N_p \setminus \{P_p\}$ if $p \in I$, and $C_p = \emptyset$ if $p \in L$.

### 2.3. Programs

The program consists of a set of *shared 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. In this paper, we consider *semi-uniform algorithms*: every processor, except one (the root), having the same degree executes the same program.

Each action is of the following form: $\langle label \rangle :: \langle guard \rangle \rightarrow \langle statement \rangle$. 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 $e = (\gamma_0, \gamma_1, \ldots, \gamma_i, \gamma_{i+1}, \ldots)$, such that for $i \geqslant 0, \gamma_i \mapsto \gamma_{i+1}$ (a single *computation step*) if $\gamma_{i+1}$ exists, or $\gamma_i$ is a terminal configuration. *Maximality* means that the sequence is either infinite or 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 $\mathcal{E}$.

A processor $p$ is said to be *enabled* in $\gamma$ ($\gamma \in \mathcal{C}$) if there exists at least an action $A$ such that the guard of $A$ is true in $\gamma$. When there is no ambiguity, we will omit $\gamma$. Similarly, an action $A$ is said to be enabled (in $\gamma$) at $p$ if the guard of $A$ is true at $p$ (in $\gamma$). We consider that any enabled processor $p$ is *neutralized* in the computation step $\gamma_i \mapsto \gamma_{i+1}$ if $p$ is enabled in $\gamma_i$ and not enabled in $\gamma_{i+1}$, but does not execute any action between these two configurations. (The neutralization of a processor represents the following situation: at least one neighbor of $p$ changes its state between $\gamma_i$ and $\gamma_{i+1}$, and this change effectively made the guard of all actions of $p$ false.) We assume an *unfair and distributed daemon*. The *unfairness* means that even if a processor $p$ is continuously enabled, then $p$ may never be chosen by the daemon unless $p$ is the only enabled processor. 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, we use the definition of *round* [11]. 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 of the protocol or the neutralization of every 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 the set $\mathcal{X}$.

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

*Specification of the DFTC protocol*: Let a predicate $Token(p)$ defined on the states of $p$ and its neighbors. A processor $p$ *holds* a token if and only if $Token(p)$ is true. A processor $p$ *sends* a token in a computation step $\alpha \mapsto \beta$ if and only if (1) $p$ holds a token in $\alpha$ and (2) $p$ executes one of some particular actions of the protocol in $\alpha \mapsto \beta$.

**Definition 2.2.** A DFTC cycle is a finite computation $e \in \mathcal{E}$ such that (1) a DFTC cycle starts when R sends one token $t$, (2) a DFTC ends when $t$ traversed the network in the depth-first search order.

**Specification 2.1** ($\mathcal{SP}_{DFTC}$). *A protocol solves the DFTC Protocol if the two following conditions are true*:

1. *If the root* R *needs to initiate a DFTC cycle, then* R *initiates it in a finite time.*
2. *If the root* R *sends a token $t$ in a computation step $\alpha \mapsto \beta$, then the prefix of every execution starting from $\alpha$ is a DFTC cycle.*

**Remark 2.1.** To prove that an algorithm provides a snap-stabilizing DFTC protocol, we need to show that both conditions of Specification 2.1 are true starting from any arbitrary configuration.

## 3. DFTC in oriented tree networks

In this section, we consider *oriented* tree networks: every processor $p$ ($p \neq$ R) knows the one of its links leads to its parent $P_p$. We first present the algorithm. Next, we prove that the algorithm is snap-stabilizing, even under an unfair daemon. The state optimality is thereafter presented.

### 3.1. The algorithm

Algorithm 3.1 describes the DFTC scheme (Algorithm $DFTC_O$). Every processor $p$ maintains a state variable $S_p$. If $p =$ R, then $S_p \in C_p \cup \{idle\}$—note that $C_p = N_p$ if $p =$ R. If $p \in I$, then $S_p \in C_p \cup \{idle, done\}$. Otherwise ($p \in L$), $S_p \in \{idle, done\}$. Note that $S_p$ can never be equal to $P_p$.

A processor $p$ is said to be *idle* ($S_p = idle$) when $p$ is ready to participate in the DFTC cycle. Similarly, a processor $p$ ($p \neq$ R) is in state *done* ($S_p = done$) when $p$ has completed the DFTC cycle, i.e., the token circulation in $T_p$ is complete. The token circulation starts from root and traverses the graph following the *First DFS Tree* rooted in R [5], i.e., visiting the adjacent edges of each processor in the order induced by $\succ_p$. A processor holds the token (predicate $token(p)$) if $forward(p)$ or $backward(p)$ is true. On every processor, the token circulation is driven by Action $O1$. The processors other than R use two extra actions, Actions $O2$ and $OC$. The purpose of Action $O2$ is to clean the trace of the token circulation after a DFTC cycle. We will show later that the algorithm performs the trace cleaning and the token circulation concurrently. Action $OC$ detects and repairs the abnormal local configurations which may exist following the initial system configuration.

**Algorithm 3.1** ($DFTC_O$) Depth-First Token Circulation In Oriented Trees.

```
Algorithm for the root R:
  Variables
    $S_p \in N_p \cup \{idle\}$
  Functions
    $Next_p \equiv (q = \min_{\succ_p}\{q' \in N_p :: q' \succ_p S_p\})$ if $q$ exists,
        $idle$ otherwise;
  Predicates
    $Forward(p) \equiv (S_p = idle)$
    $Backward(p) \equiv (\exists q \in N_p :: S_p = q \wedge S_q = done)$
    $Token(p) \equiv Forward(p) \vee Backward(p)$
    $Permission(p) \equiv (Next_p = idle \vee S_{Next_p} = idle)$
  Actions
    $O1 :: Token(p) \wedge Permission(p) \rightarrow S_p := Next_p$;
.....................................................................
Algorithm for the other nodes ($p \neq$ R):
  Variables
    $S_p \in C_p \cup \{idle, done\}$
  Functions
    $Next_p \equiv (q = \min_{\succ_p}\{q' \in C_p :: q' \succ_p S_p\})$ if $q$ exists, $done$ otherwise;
  Predicates
    $Forward(p) \equiv (S_{P_p} = p) \wedge (S_p = idle)$
    $Backward(p) \equiv (S_{P_p} = p) \wedge (\exists q \in C_p :: S_p = q \wedge S_q = done)$
    $Token(p) \equiv Forward(p) \vee Backward(p)$
    $Permission(p) \equiv (Next_p = done \vee S_{Next_p} = idle)$
    $Clean(p) \equiv (S_p = done) \wedge (S_{P_p} \neq p)$
    $AbnRoot(p) \equiv (S_{P_p} \neq p) \wedge (S_p \in C_p)$
  Actions
    $O1 :: Token(p) \wedge Permission(p) \rightarrow S_p := Next_p$;
    $O2 :: Clean(p) \rightarrow S_p := idle$;
    $OC :: AbnRoot(p) \rightarrow S_p := idle$;
```

Let us explain first the normal behavior of Algorithm $DFTC_O$ following the example shown in Fig. 1. (We will consider how Action $OC$ deals with the abnormal configurations thereafter.) Although our algorithm is designed to work under any asynchronous daemon, to simplify the explanation of the example, we assume that in every configuration the daemon chooses all enabled processors to execute an action. (We will remove this restriction later.)

Let us assume the system starts in the configuration where all processors are in state *idle* (Configuration (i) in Fig. 1). In such a configuration, only the root R is enabled (with Action $O1$). This means that the root is enabled to start a DFTC cycle. Then, the root chooses a neighbor as its *successor* (Processor $a$ in Fig. 1), i.e., this successor in the next processor that will have the token. This is shown in Configuration (ii). Similarly, Processor $a$ chooses a successor (Configuration (iii)) executing Action $O1$. Processor $b$ does not have any child to choose from. So, $b$ executes $S_b := done$ (see Macro *Next* and Configuration (iv)). $Backward(a)$ becomes true leading $a$ to choose Processor $c$ as successor (Configuration (v)).

Concurrently, Action $O2$ becomes enabled on Processor $b$ because $b$ is *done* and it is not the current successor of Processor $a$ (see Predicate $Clean(p)$). Concurrent executions of both $O1$ (on $c$) and $O2$ (on $b$) leads the system in Configuration (vi). This indicates to Processor $a$ that the token has traversed all processors in $T_a$, the subtree rooted at $a$. Next, Actions $O1$ and $O2$ are repeated concurrently until all processors are visited by the token (Configurations (vii) to (x)).

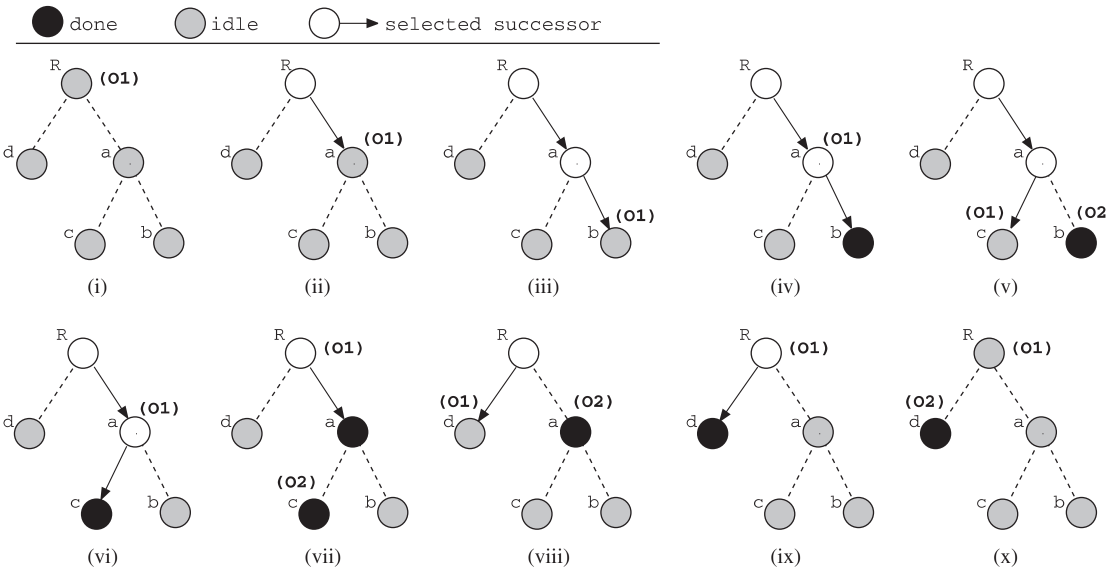

Fig. 1. An example showing the normal behavior of Algorithm 3.1.

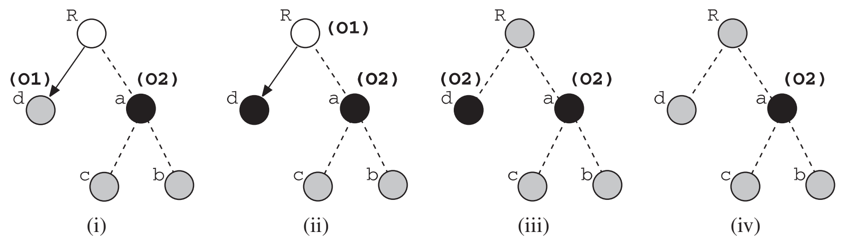

Fig. 2. An example showing how two consecutive DFTC Cycles are prevented to mix together.

Clearly, the configuration following Configuration (x) is either Configuration (i)—in this case, the root R does not start another DFTC cycle—or Configuration (ii)—R needs to start a new DFTC cycle.

Now, let us remove the restriction we imposed to the daemon in the example of execution we described in Fig. 1, i.e., we now consider the asynchronicity property of the distributed daemon again. So, some processors may be slow to execute Action $O2$. Assuming repeated executions of the DFTC cycle, some (slow) processors may be involved in a DFTC cycle whereas some other are still *done* of the previous cycle. For instance, consider the execution shown in Fig. 1 again. From Configuration (viii) and the following ones, Processor $a$ could be slow to execute Action $O2$ while the token keeps circulating. This could make the execution as shown in Fig. 2 leading the system in Configuration (iv) where the root should be able to initiate a new DFTC Cycle—i.e., as in Configuration (i) of Fig. 1. So, without taking any kind of precaution, two consecutive DFTC cycles could mix together.

To avoid this kind of problem, a processor $p$ is allowed to select $q$ as its successor ($q \in C_p$) once $S_q = idle$ (i.e., $q$ executed Action $O2$). That is the aim of Predicate *Permission*$(p)$. Configuration (iv) in Fig. 1, Action $O1$ is not enabled on R because *Permission*(R) is false. After Processor (a) executed Action $O2$, Action $O1$ becomes enabled on R. Thus, the processors which are slow to execute their Action $O2$, are protected from the next DFTC cycle.

Since the systems we consider require no initialization, they can start from any configuration. Informally, a configuration is an "*abnormal*" configuration if there exists at least an internal processor $p$ ($p \in I$) and there exists $q \in C_p$ such that $S_p = q$, and $S_{P_p} \neq p$. We call such a processor an *abnormal root*. In Fig. 3, the processors $p$ and $q$ are abnormal roots. In such a configuration, Predicate *AbnRoot*$(p)$ is true, and $p$ eventually executes Action *OC*. By induction, every abnormal root eventually disappears in the worst case once it reaches a leaf processor by successive execution of Action *OC*. It is easy to see that the removal of the abnormal roots does not interfere with the DFTC cycle (rooted in R). Thus, we can prove that Algorithm $DFTC_O$ is snap-stabilizing.

### 3.2. *Proof of snap-stabilization*

**Definition 3.1** (*Path*). A sequence of processors $p_0, \ldots, p_k$ is called a path if and only if $k>0 \Rightarrow (\forall i,\ 1\leqslant i\leqslant k,\ (p_i \in N_{p_{i-1}} \wedge (\forall j,\ 0\leqslant j<i,\ p_j \neq p_i))$. $k$ is called the distance from $p_0$ to $p_k$, denoted $d(p_0, p_k)$.

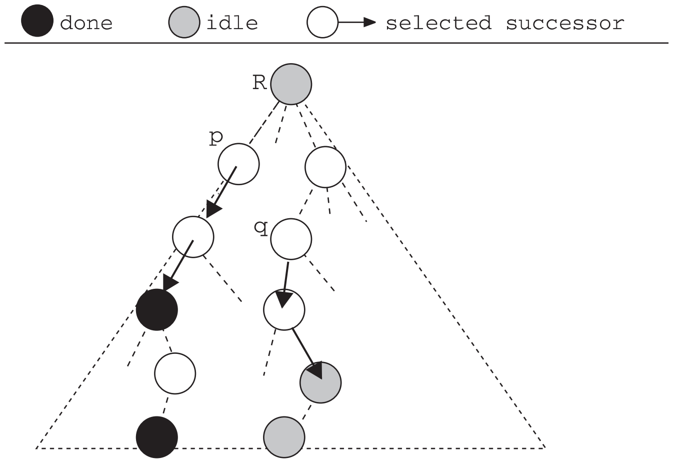

Fig. 3. An example showing an abnormal initial configuration.

We use the notation $(p,\alpha)\vdash Token$ to indicate that $Token(p)$ is true in Configuration $\alpha$. When there is no ambiguity, we will simply write $Token(p)$.

**Definition 3.2** (*Active path*). For any node $p \in \{\mathtt{R}\} \cup I$, for any configuration $\alpha \in \mathcal{C}$, the *active path* $\overrightarrow{p}$ is the unique path $p_0, \ldots, p_k$ such that (i) $p = p_0$, (ii) $k > 0 \Rightarrow (\forall i,\ 0 \leqslant i < k,\ S_{p_i} = p_{i+1})$, and (iii) $(p_k,\alpha)\vdash$Token. $\forall 0 \leqslant i \leqslant k$, $p_i$ is said to belong to $\overrightarrow{p}$ in $\alpha$ and is denoted as $p_i \in_\alpha \overrightarrow{p}$. When there is no ambiguity, we will omit $\alpha$.

We use the notation $(p,\alpha)\vdash AR$ to indicate that $AbnRoot(p)$ $(p \in I)$ is true in configuration $\alpha$. $(p,\alpha)\nvdash AR$ denotes that $AbnRoot(p)$ is false in $\alpha$.

In the remainder, $\overrightarrow{\mathtt{R}}$ is called the *normal* (active) path. Every active path $\overrightarrow{p}$ such that $p\vdash AR$ is called an *abnormal rooted* (active) path.

The following lemma shows that a processor can become an abnormal root only when a neighboring abnormal root is pushed down towards one of its children. In other words, that means that, following any arbitrary initial configuration, the number of abnormal roots never increases and if some abnormal roots exists, then each of them is pushed toward the leaf processors.

**Lemma 3.1.** $\forall p \in I, \forall \alpha, \beta \in \mathcal{C} : \alpha \mapsto \beta :: (p,\alpha)\nvdash AR \wedge (p,\beta)\vdash AR \Rightarrow (P_p \in I) \wedge (S_{P_p} = p \textit{ in } \alpha) \wedge ((P_p,\alpha)\vdash AR) \wedge ((P_p,\beta)\nvdash AR)$.

**Proof.** Since, $(p,\alpha)\nvdash AR$, we have to consider three possible situations in $\alpha$: (1) $S_p = idle$, (2) $(\exists q \in C_p ::\ S_p = q) \wedge (S_{P_p} = p)$, and (3) $S_p = done$ in $\alpha$. If $S_p = done$, then no action can change $S_p$ such that $S_p = q$ $(q \in C_p)$ will be true in $\beta$. Thus, this case does not satisfy the assumptions made in the lemma statement. So, we consider the two first cases only.

Assume first that $S_p = idle$ in $\alpha$. Since $(p,\beta)\vdash AR$, $p$ executes an action from $\alpha$ such that $S_p$ changes from *idle* in $\alpha$ to $q$ $(q \in C_p)$ in $\beta$. The only possible action that can make this happen is Action $O1$. This implies that $S_{P_p} = p$ in $\alpha$, and $P_p$ executes an action in $\alpha$ such that $S_{P_p} \neq p$ in $\beta$. The only possible action which $P_p$ can execute to make this change is Action $OC$. Hence, $P_p \in I$ and $(P_p,\alpha)\vdash AR$. After $P_p$ executes $OC$ from $\alpha$, $S_{P_p} = idle$ becomes true in $\beta$. So, $(P_p,\beta)\nvdash AR$.

Secondly, assume that $(\exists q \in C_p ::\ S_p = q) \wedge (S_{P_p} = p)$ in $\alpha$. Then, $P_p$ cannot execute Action $O1$ from $\alpha$. So, the only possible action $P_p$ can execute from $\alpha$ is Action $OC$. Hence, as in Case 1, $P_p \in I$ and $(P_p,\alpha)\vdash AR$ and $(P_p,\beta)\nvdash AR$. □

**Lemma 3.2.** $\forall p \in V : \forall \alpha \in \mathcal{C}, \forall e \in \mathcal{E}_\alpha :: p$ *cannot execute Action OC infinitely often.*

**Proof.** Assume by contradiction that $p$, $\alpha$, and $e$ exist such that $p$ executes Action $OC$ infinitely often. By Lemma 3.1, the assumption is also true for $P_p$. By induction, this implies that $q$, the neighbor of R on the path from R to $p$, executes Action $OC$ infinitely often. By Lemma 3.1 again, $P_q \in I$, which implies that $P_q$ cannot be equal to R. A contradiction. □

**Definition 3.3.** A processor $p$ is said to be Locked from a configuration $\alpha$, denoted $(p,\alpha)\vdash$Locked, if there exists an execution $e$ starting from $\alpha$ $(e \in \mathcal{E}_\alpha)$, such that $p$ executes no action.

Let $\mathcal{E}_\alpha^{\mathrm{Lock}(p)}$ be the subset of $\mathcal{E}_\alpha$ where $p$ executes no action. Note that $p$ being locked does not imply that $p$ is not enabled any more. This means that, even if $p$ is infinitely often enabled, the daemon never chooses $p$ to execute an action.

We now show that if a processor is locked, then all its neighbors are eventually locked (Lemma 3.3). We first need the following lemma:

**Lemma 3.3.** $\forall p \in V : \forall \gamma_0 \in \mathcal{C} :: (p,\gamma_0)\vdash Locked \Rightarrow (\forall e = (\gamma_0,\gamma_1,\ldots) \in \mathcal{E}_{\gamma_0}^{\mathrm{Lock}(p)}, \exists i \geqslant 0 ::\ \forall q \in N_p, (q,\gamma_i)\vdash Locked)$.

**Proof.** There are three cases to consider, depending on $p = \mathtt{R}$, $p \in L$, or $p \in I$.

Let us assume first that $p \in I$. We prove the result by contradiction. Assume that there exists a processor $q \in N_p$ and an execution $e \in \mathcal{E}_{\gamma_0}^{\mathrm{Lock}(p)}$ such that $q$ moves infinitely often in $e$. By Lemma 3.2, $q$ cannot execute Action $OC$ infinitely often. So, eventually, $q$ executes only $O1$ or $O2$. We now consider that $q$ never executes $OC$. Function $Next_q$ is based on $\succ_q$ which is a strict order. So, infinitely often, $q$ repeats the sequence of actions: $|C_q|$ successive executions of $O1$, followed by an execution of Action $O2$ if $q \neq \mathtt{R}$. There are two cases to consider:

1. Assume $q = P_p$. In that case, in any infinite executions of $O1$ (and $O2$) by $q$, $Next_q$ eventually returns $p$. If $S_p \neq idle$, then $Permission(q)$ will be never true (since $p$ is locked). If $S_p = idle$, then once $S_q = p$, $q$ is not enabled forever. In both cases, $q$ is locked. A contradiction.
2. Assume $q \in C_p$. Assume first that $S_p \neq q$ in $\gamma_0$. If $S_q = idle$ then $q$ is not enabled forever. So, $q$ is Locked.

A contradiction. If $S_q = done$, then Action $O2$ is the only possible action that $q$ can execute. After the execution of Action $O2$, $S_q = idle$. A contradiction. If $S_q \in C_q$, then, the only action $q$ can execute is $OC$ which is not possible by assumption.
Now, assume that $S_p = q$ in $\gamma_0$. Since $C_q$ is finite, $S_q$ is eventually equal to *done* and $q$ is not enabled forever. So, $q$ is eventually locked. A contradiction.

The proof for the case $p \in L$ is similar to Case 1. The proof for the case $p = \mathtt{R}$ is similar to Case 2. □

By induction of Lemma 3.3, the following corollary holds:

**Corollary 3.1.** $\exists X \subseteq V, X \neq \emptyset, \gamma_0 \in \mathcal{C} :: (\forall p \in X, (p, \gamma_0) \vdash Locked) \Rightarrow (\forall e = (\gamma_0, \gamma_1, \ldots) \in \mathcal{E}_{\gamma_0}^{\mathrm{Lock}(p)}, \exists i \geqslant 0 :: (\forall q \in V, (q, \gamma_i) \vdash Locked))$.

Corollary 3.1 implies that the daemon cannot prevent any continuously enabled processor to move forever. So, we can claim the following theorem:

**Theorem 3.1.** *Any round of Algorithm* $DFTC_O$ *is finite.*

In the remainder, we distinguish the processor which holds the token at the extremity of the active path rooted in R, i.e., the "normal" token—by opposition of the "abnormal" tokens which are at the extremity of active paths rooted in abnormal roots. More formally, considering any configuration $\gamma \in \mathcal{C}$, we define Predicate $AP$ as follows:

$$(p, \alpha) \vdash AP \overset{\mathrm{def}}{\equiv} (p, \alpha) \vdash Token \wedge (p \in \overrightarrow{\mathtt{R}} \text{ in Configuration } \alpha).$$

By convenience, in the remainder, when there is no ambiguity, if $(p, \alpha) \vdash AP$ holds, then $p$ is called the (*normal*) *active processor* (in $\alpha$), simply denoted $AP$. We now show that processors which are either done or abnormal roots cannot prevent the "normal" token ($AP$) progress (Lemma 3.6). This result leads to the fact that, starting from any configuration, the "normal" token traverses the tree network in a time closed to the expected time to traverse a tree in the depth-first search order, except one round in the general case—two if the root has only one neighbor (Theorem 3.2). This shows the final result, i.e., Algorithm $DFTC_O$ is snap-stabilizing (Theorem 3.3).

**Lemma 3.4.** $\forall \gamma_0 \in \mathcal{C},\ \forall e \in \mathcal{E}_{\gamma_0}, \forall p \in I :: (p, \gamma_0) \vdash AR \Rightarrow p$ *executes Action OC in at most one round.*

**Proof.** From Algorithm 3.1, no processor $p$ ($\in I$) such that $AbnRoot(p)$ is true can be chosen executing Action $O1$. So, $p$ is continuously enabled until $p$ executes Action $OC$. By Theorem 3.1, $p$ executes Action $OC$ in at most one round. □

**Lemma 3.5.** $\forall \gamma_0 \in \mathcal{C},\ \forall e \in \mathcal{E}_{\gamma_0}, \forall p \in I \cup L :: (S_p = done) \wedge (S_{P_p} \neq p)$ *in* $\gamma_0 \Rightarrow p$ *executes Action* $O2$ *in at most one round.*

**Proof.** The proof is similar to the proof of Lemma 3.4. □

Let *NVC* (*NVC* stands for *Non-Visited Children*) be the subset of $C_{AP}$ defined as follow:

$$NVC = \begin{cases} C_{AP} & \text{if } S_{AP} = idle, \\ \{p \in C_{AP} :: \ p \succ_{AP} S_{AP}\} & \text{otherwise.} \end{cases}$$

**Lemma 3.6.** $\forall \gamma_0 \in \mathcal{C},\ \forall e \in \mathcal{E}_{\gamma_0}$ :: *if AP executes less than k actions in k rounds, then* $\forall p \in NVC :: \ S_p = idle$ *at the beginning of round* $k + 1$.

**Proof.** Assume by contradiction that $\exists \gamma_0 \in \mathcal{C},\ \exists e \in \mathcal{E}_{\gamma_0}$ such that $AP$ executes less than $k$ actions in $k$ rounds and $\exists p \in NVC :: \ S_p \in \{done, C_p\}$ at the beginning of round $k+1$. Let $k$ be the minimal value where the above assumption is true, i.e., $\forall j \in 0 \leqslant j \leqslant k-1$, the number of AP's actions in $j$ rounds is greater than or equal to $j$. Since $k$ is minimal, the number of actions executed by $AP$ is equal to $k-1$, i.e., $AP$ executes no action during the $k$th rounds. If $S_p \in C_p$ during the $k$th round ($p$ is an abnormal root), then by Lemma 3.4, $S_p = idle$ at the beginning of round $k+1$. If $S_p = done$ during the $k$th round, then by Lemma 3.5, $S_p = idle$ at the beginning of round $k+1$. As a result, $\forall p \in NVC :: \ S_p = idle$ at the beginning of round $k+1$, which contradicts the assumption $\exists p \in NVC :: \ S_p \in \{done, C_p\}$ at the beginning of round $k+1$. □

**Lemma 3.7.** $\forall \gamma_0 \in \mathcal{C},\ \forall e \in \mathcal{E}_{\gamma_0}$ :: *At the end of any round* $i$ ($i \geqslant 1$), *the number of actions executed by AP is greater than or equal to* $i - 1$.

**Proof.** The lemma is obvious if $i = 1$. Let us prove by contradiction the case where $i > 1$. Assume that $\exists \gamma_0 \in \mathcal{C},\ \exists e \in \mathcal{E}_{\gamma_0}$ such that there exists $k \geqslant 1$ where the number of actions executed by $AP$ is less than $k-1$ at the end of the $k$th round. Let $k$ be the minimal value. So, during the first $k-1$ rounds, $AP$ executed $k-2$ actions (otherwise $k$ is not minimal) and $AP$ executes no action during the $k$th round. By Lemma 3.6, $\forall p \in NVC :: \ S_p = idle$ at the beginning of the $k$th round. Thus, $AP$ is enabled at the beginning of this round (with Action $O1$). By Theorem 3.1, $AP$ executes Action $O1$ during the $k$th round. A contradiction. □

Let us call *SC* ("Starting Configuration"), the configuration set where the root is enabled ($S_{\mathtt{R}} = idle$ and $S_{Next_{\mathtt{R}}} = idle$). The next theorem shows that starting from any configuration, the system eventually reaches a starting configuration ($\in SC$) in finite time.

**Theorem 3.2.** $\forall \gamma_0 \in \mathcal{C},\ \forall e = (\gamma_0, \gamma_1, \ldots) \in \mathcal{E}_{\gamma_0}$ :: *there exists* $i \geqslant 0$ *and* $k \geqslant 1$ *such that*, (1) $\gamma_i$ *belongs to the kth round*, (2) $\gamma_i \in SC$, *and* (3) $k \leqslant 2n-1$ *if* $|N_{\mathtt{R}}| > 1$, $k \leqslant 2n$ *otherwise* (R *has a single neighbor*).

**Proof.** We first compute $k'$, the number of actions executed by $AP$ to reach a configuration where $S_{\mathtt{R}} = idle$, starting from a configuration in $SC$. Action $O1$ must be executed at least $2(n-1)$ times (the token traverses each edge twice). Once the last neighbor $p$ of R returns the token to R ($S_{\mathtt{R}}=p$

and $S_p = done$), R must execute Action $O1$ once more to become *idle* again. So, $k' = 2n - 1$. From the above discussion, the number of actions, executed by $AP$ to reach a configuration where $S_R = idle$ starting from a configuration which is not in $SC$, is $k' - 1$. By Lemma 3.7, the maximum number of rounds is less than $k' + 1$, hence, exactly $k'$ rounds. Thus, if $p$ is the unique neighbor of R, then $p$ must execute Action $O2$ to clean its state from *done* to *idle*, which leads to a configuration in $SC$. (Note that if $p$ is not the unique neighbor of R, then the previous configuration is already in $SC$). Thus, $k = k'$ if $|N_R| > 1$, $k=k' + 1$ otherwise. □

From Theorem 3.2, if the root needs to send the token, its sends it after at most $2n - 1$ rounds. Furthermore, starting from a configuration where R is enabled ($S_R = idle$ and $S_{Next_R} = idle$), the token traverses the tree network following the depth-first search order induced by $\succ_p$ ($\forall p \in V$). So, by Remark 2.1, we can claim:

**Theorem 3.3** (*snap-stabilizing*). *Algorithm $DFTC_O$ is a snap-stabilizing DFTC algorithm.*

### 3.3. State optimality

The following theorem is proven in [19].

**Theorem 3.4** (*Petit and Villain [19]*). *To implement a DFTC cycle on an oriented rooted tree network, the number of states per processor p must be greater than or equal to $\Delta_p + 1$.*

Clearly, Algorithm $DFTC_O$ requires only $\Delta_p + 1$ states per processor $p$ (for its only variable, $S_p$). So, from this, Theorem 3.3, and Theorem 3.4, we can claim the following result:

**Theorem 3.5.** *Algorithm $DFTC_O$ is optimal in terms of the number of states per processor to implement a DFTC cycle on an rooted oriented tree network.*

## 4. DFTC in un-oriented trees

In this subsection, we consider un-oriented rooted tree networks, i.e., no processor $p$ ($p \neq$ R) knows the one of its links leads to its parent $P_p$. $C_p$, the set of topological children cannot be use as in Section 3. So, we are using $N_p$ instead of $C_p$ in the code (for every processor which is not the root R). In the next subsection, we present the algorithm, followed by the proof of snap-stabilizing (under an unfair daemon). The state optimality is thereafter presented.

### 4.1. The algorithm

Algorithm 4.2 describes the DFTC scheme in un-oriented trees.

The normal DFTC cycle behaves as in Algorithm $DFTC_O$. Every processor $p \in I$ maintains an additional state allowing $p$ to have $P_p$ as successor. This is due to the fact that no processor in $I$ knows which of its neighbor is its parent $P_p$ (the tree is not oriented). So, in an abnormal configuration, a processor $p \in I$ can have $P_p$ as successor, every processor $p$ (including R) can be the successor of several of its neighbors and, *cycles* may exist, i.e., pairs of neighbors $\{p, q\}$ such that $q$ is the current successor of $p$ while $p$ is the successor of $q$ ($S_p = q \wedge S_q = p$).

In Algorithm 4.2, every processor $p \neq$ R locally refers to the set of its predecessor as $Pred_p$. Every cycle $\{p, q\}$ is *free* if $Pred_p = \{q\} \wedge Pred_q = \{p\}$, otherwise $\{p, q\}$ is *linked*. Moreover, a linked cycle $\{p, q\}$ is *2-linked* if $|Pred_p| > 1$ and $|Pred_q| > 1$, i.e., both $p$ and $q$ have a predecessor which is not into the cycle. A linked cycle not being 2-linked is *1-linked*. Examples of such cycles are shown in Fig. 4.

**Algorithm 4.2** (*$DFTC_U$*) DFTC In Un-Oriented Trees.

```
Algorithm for the root R:
  Variables
    $S_p \in N_p \cup \{idle\}$
  Functions
    $Next_p \equiv (q = \min_{\succ_p}\{q' \in N_p :: q' \succ_p S_p\})$ if $q$ exists,
          $idle$ otherwise;
  Predicates
    $Forward(p) \equiv (S_p = idle)$
    $Backward(p) \equiv (\exists q \in N_p :: S_p = q \wedge S_q = done)$
    $Token(p) \equiv Forward(p) \vee Backward(p)$
    $Permission(p) \equiv (Next_p = idle \vee S_{Next_p} = idle)$
  Actions
    $O1 :: Token(p) \wedge Permission(p) \rightarrow S_p := Next_p$;

Algorithm for the other nodes ($p \neq$ R):
  Variables
    $S_p \in N_p \cup \{idle, done\}$ if $p \in I$,
    $S_p \in \{idle, done\}$          if $p \in L$
  Macros
    $Pred_p \equiv \{q \in N_p :: S_q = p\}$
  Functions
    $Next_p \equiv (q = \min_{\succ_p}\{q' \in N_p \setminus Pred_p :: q' \succ_p S_p\})$ if $q$ exists,
          $done$ otherwise;
  Predicates
    $Forward(p) \equiv (|Pred_p| = 1) \wedge (S_p = idle)$
    $Backward(p) \equiv (|Pred_p| = 1) \wedge (\exists q \in N_p :: S_p = q \wedge S_q = done)$
    $Token(p) \equiv Forward(p) \vee Backward(p)$
    $Permission(p) \equiv (Next_p = done \vee S_{Next_p} = idle)$
    $Clean(p) \equiv (|Pred_p| = 0) \wedge (S_p = done)$
    $AbnRoot(p) \equiv (|Pred_p| = 0) \wedge (S_p \in N_p)$
    $LFCycle(p) \equiv (|Pred_p| = 1) \wedge (S_p \in Pred_p)$
  Actions
    $O1 :: Token(p) \wedge Permission(p) \rightarrow S_p := Next_p$;
    $O2 :: Clean(p)                      \rightarrow S_p := idle$;
    $OC :: AbnRoot(p) \vee LFCycle(p)    \rightarrow S_p := idle$;
```

To be snap-stabilizing, Algorithm 4.2 must remove all abnormal local configurations without interfering with the “normal” DFTC cycle, i.e., the one initiated by R. That is the aim of Action $OC$. With that action, the abnormal roots are removed similarly as in Algorithm $DFTC_O$, i.e., every illegal root is shifted until the sequence of successors starting from it eventually disappears.

Since, every abnormal root eventually disappears, every 2-linked cycle eventually becomes either free or 1-linked (at least one abnormal root is removed), even if it is created by two neighbors holding a token, and choosing each other as successor (at least one abnormal root is removed). Algorithm 4.2

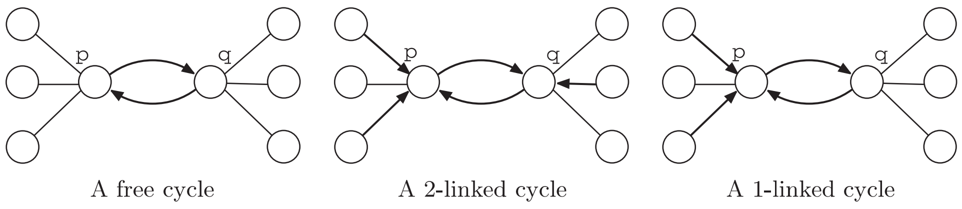

Fig. 4. Abnormal local configurations.

removes cycles once they are free or 1-linked: A processor $p$ being *locally free cycle* ($Pred_p = \{q\}$ such that $S_p = q$, see $LFCycle(p)$ in Algorithm 4.2) becomes *idle* executing Action *OC*. If the sequence of successors starting from the root R ends with a cycle, then that cycle is eventually 1-linked because the abnormal roots linked to that cycle eventually disappear. In that case, after the execution of Action *OC*, it is easy to observe that the normal DFTC cycle behaves correctly, i.e., as if the abnormal configurations never existed.

### *4.2. Proof of snap-stabilization*

In the following, we keep the definitions and notations used in Section 3, except the definition of an active path, adapted to take in account the possibility of cycles.

We need the notation $(\{p, q\}, \gamma)\vdash\{Free, 1C_p, 1C_q, 2C\}$-*Cycle* to indicate that, in configuration $\gamma$, the cycle $\{p, q\}$ is, respectively:

1. A free cycle ($Pred_p = \{q\} \wedge Pred_q = \{p\}$),
2. A 1-linked cycle, linked on $p$'s side ($Pred_p \supsetneq \{q\} \wedge Pred_q = \{p\}$),
3. A 1-linked cycle, linked on $q$'s side ($Pred_p = \{q\} \wedge Pred_q \supsetneq \{p\}$),
4. A 2-linked cycle ($Pred_p \supsetneq \{q\} \wedge Pred_q \supsetneq \{p\}$).

When it is not necessary, we will omit the prefix and simply write $(\{p, q\}, \gamma)\vdash$*Cycle*. When there is no ambiguity, we will omit $\gamma$ and simply write $\{p, q\}\vdash$*Cycle*.

**Definition 4.1** (*Active path*). For any node $p \in \{\mathtt{R}\} \cup I$, for any configuration $\alpha \in \mathcal{C}$, the *active path* $\overrightarrow{p}$ is the unique path $p_0, \ldots, p_k$ such that (i) $p = p_0$, (ii) $k > 0 \Rightarrow (\forall i,\ 0 \leqslant i < k,\ S_{p_i} = p_{i+1})$, and (iii) $(p_k, \alpha)\vdash\text{Token} \vee (\{p_{k-1}, p_k\}, \alpha)\vdash\text{Cycle}$. $\forall i \geqslant 0$, $p_i$ is said to belong to $\overrightarrow{p}$ in $\alpha$ and is denoted as $p_i \in_\alpha \overrightarrow{p}$. When there is no ambiguity, we will omit $\alpha$.

**Remark 4.1.** For every active path $\overrightarrow{p} = p_1, \ldots, p_k$ with $k \geqslant 2$, if there exists $i$, $1 \leqslant i < k$ such that $\{p_i, p_{i+1}\}\vdash$*Cycle*, then $\{p_i, p_{i+1}\}$ is the unique cycle of $\overrightarrow{p}$ and $\{p_i, p_{i+1}\} = \{p_{k-1}, p_k\}$.

We will now show that all abnormal local configurations eventually disappear. The following lemma shows that a processor $p$ can become an abnormal root only when either every predecessor of $p$ is an abnormal root, or $p$ is in a free cycle or in a 1-linked cycle such that every processor not involved in the cycle is an abnormal root.

**Lemma 4.1.** $\forall p \neq r : \forall \alpha, \beta \in \mathcal{C} : \alpha \mapsto \beta :: (p, \alpha)\nvdash AbnRoot \wedge (p, \beta)\vdash AbnRoot \Rightarrow (Pred_p \neq \emptyset\ in\ \alpha) \wedge (\forall q \in Pred_p\ in\ \alpha : (S_q = idle\ in\ \beta) \wedge ((q, \alpha)\vdash AbnRoot \vee (\{p, q\}, \alpha)\vdash Free\text{-}Cycle \vee (\{p, q\}, \alpha)\vdash 1C_q\text{-}Cycle))$.

**Proof.** We prove this by contradiction. First of all, $p$ cannot be *done* in $\alpha$ because no action allows $p$ to be an abnormal root in $\beta$ ($p$ can execute $O2$ only).

By contradiction, we consider the following cases:

1. Assume that $Pred_p = \emptyset$ in $\alpha$. Then, $p$ can execute Action *OC* only. So, $(p, \beta)\nvdash AbnRoot$ which contradicts the assumption.
2. Assume that $\exists q \in Pred_p$ in $\alpha : S_q \neq idle$ in $\beta$. So, $S_q = done$ or $S_q = p'$ in $\beta$ ($p' \in N_q$). In both cases, that implies that $q$ executed $O1$ in $\alpha$ and, then, $S_p = done$ in $\alpha$, which is impossible.
3. Assume that $\exists q \in Pred_p$ in $\alpha$ :: $(q, \alpha)\nvdash AbnRoot \wedge (\{p, q\}, \alpha)\nvdash Free\text{-}Cycle \wedge (\{p, q\}, \alpha)\nvdash 1C_q\text{-}Cycle$. So, $Pred_q \neq \emptyset$ and either $p \notin Pred_q$, or $(\{p, q\}, \alpha)\vdash 2C\text{-}Cycle$. In that cases, $q$ can execute an action only if $S_p = done$, which is impossible. □

The following lemma shows that if a cycle is created, then exactly two rooted paths (so, at least one abnormal rooted path) are involved.

**Lemma 4.2.** $\forall\{p, q\} : \forall \alpha, \beta \in \mathcal{C} : \alpha \mapsto \beta :: (\{p, q\}, \alpha)\nvdash Cycle \wedge (\{p, q\}, \beta)\vdash Cycle \Rightarrow ((S_p = idle\ in\ \alpha) \wedge (S_q = idle\ in\ \alpha) \wedge (\exists! q' \in N_p \setminus \{q\},\ \exists! p' \in N_q \setminus \{p\} ::\ S_{q'} = p \wedge S_{p'} = q\ in\ \alpha))$.

**Proof.** Only Action $O1$ allows $p$ to change $S_p$ to $q$ during $\alpha \mapsto \beta$. So, $|Pred_p| = 1$ and $S_q = idle$ in $\alpha$. The configuration is symmetric for $q$. □

As in the previous section, we show that every round is finite (Lemma 4.3–Corollary 4.2).

**Lemma 4.3.** $\forall p \in V : \forall \gamma_0 \in \mathcal{C}, \forall e \in \mathcal{E}_{\gamma_0}$ :: *$p$ cannot execute Action OC infinitely often.*

**Proof.** Assume by contradiction that $\exists p \in V : \exists \gamma_0 \in \mathcal{C}, \exists e \in \mathcal{E}_{\gamma_0}$ :: $p$ executes Action *OC* infinitely often. Consider the following cases:

1. $p$ belongs a finite number of times to a cycle. So, eventually, each time *OC* is enabled, $AbnRoot(p)$ is true. We now consider a suffix $e'$ of $e$ such that $p$ never belongs to a cycle. Since the number of neighbors of $p$ is finite, there exists a

neighbor $q$ such that $q \in Pred_p$ infinitely often. By Remark 4.1, $q$ cannot belong to a cycle. By Lemma 4.1, $AbnRoot(q)$ holds infinitely often. So, $q$ executes Action $OC$ infinitely often. Since $p$ and $q$ execute infinitely often $OC$ in the same conditions, there exists infinitely often an active path $\overrightarrow{p'}$ such that (i) $p \in \overrightarrow{p'}$ and (ii) either $p' = \mathtt{R}$ or $p' \in L$ and $p'$ executes Action $OC$ infinitely often. But $p'$ cannot be equal to $\mathtt{R}$ because Action $OC$ is not an action on $\mathtt{R}$. Similarly, no leaf processor can take its unique neighbor as successor ($\forall p' \in L,\ S_{p'} \in \{idle, done\}$). So, this case is impossible.
2. $p$ belongs to cycle ($\{p, q\}$) infinitely often. So, by Lemma 4.2 and Remark 4.1, there exists $p' \in N_p \setminus \{q\}$ such that $S_{p'} = p$ infinitely often. Following the same reasoning on $p'$ as in Case 1 for processor $q$ leads to the same conclusion. □

**Lemma 4.4.** $\forall p \in V : \forall \alpha \in \mathcal{C} :: (p,\alpha)\vdash Locked \Rightarrow (\forall e = (\gamma_0, \gamma_1, \ldots) \in \mathcal{E}_{\alpha}^{\mathrm{Lock}(p)}, \exists i \geqslant 0 :: \forall q \in N_p,\ (q, \gamma_i)\vdash Locked)$.

**Proof.** We prove the result by contradiction. Assume that there exists a processor $q \in N_p$ and an execution $e \in \mathcal{E}_{\gamma_0}^{\mathrm{Lock}(p)}$ such that $q$ moves infinitely often in $e$. By Lemma 4.3, $q$ cannot execute Action $OC$ infinitely often. So, eventually, $q$ executes only $O1$ or $O2$. We now consider that $q$ never executes $OC$. Function $Next_q$ is based on $\succ_q$ which is a strict order. So, infinitely often, $q$ repeats the sequence of actions: $|C_q|$ successive executions of $O1$, followed by an execution of Action $O2$ if $q \neq \mathtt{R}$. There are two cases to consider:

1. Assume $S_p \neq q$. In that case, in any infinite executions of $O1$ (and $O2$) by $q$, $Next_q$ eventually returns $p$. If $S_p \neq idle$, then $Permission(q)$ will be never true (since $p$ is locked). If $S_p = idle$, then once $S_q = p$, $q$ is not enabled forever. In both cases, $q$ is locked. A contradiction.
2. Assume $S_p = q$. After at most $|C_q|$ executions of $O1$, $S_q = done$ and $q$ is locked. A contradiction. □

**Corollary 4.1.** $\exists X \subseteq V, X \neq \emptyset, \gamma_0 \in \mathcal{C} :: (\forall p \in X, (p, \gamma_0)\vdash Locked) \Rightarrow (\forall e{=}(\gamma_0, \gamma_1, \ldots){\in}\mathcal{E}_{\gamma_0}^{\mathrm{Lock}(p)}, \exists i \geqslant 0{::} (\forall q{\in}V, (q, \gamma_i)\vdash Locked))$

**Corollary 4.2.** *Any round of Algorithm $DFTC_U$ is finite.*

Now, as in Section 3, we will show that Algorithm $DFTC_U$ is snap-stabilizing.

**Lemma 4.5.** $\forall \gamma_0 \in \mathcal{C},\ \forall e \in \mathcal{E}_{\gamma_0}, \forall p \in I$ :: *if Action OC is enabled on p, then p executes Action OC in at most one round.*

**Proof.** From Algorithm 4.2, no processor $p$ ($\in I$) such that $AbnRoot(p)$ or $LFCycle(p)$ is true can be chosen by one of its neighbors executing Action $O1$. So, $p$ is continuously enabled until $p$ executes Action $OC$. By Corollary 4.2, $p$ executes Action $OC$ in at most one round. □

Considering any configuration $\gamma \in \mathcal{C}$, we define the set $DT$ ("down–top") as the set of processors such that $\forall p \in DT :: S_p = P_p$.

**Lemma 4.6.** $\forall \gamma_0 \in \mathcal{C},\ \forall e = (\gamma_0, \gamma_1, \ldots) \in \mathcal{E}_{\gamma_0}$ :: *there exists $i \geqslant 0$ and $k$, $1 < k < h$ such that,* (1) *$\gamma_i$ belongs to the kth round,* (2) *$\forall j \geqslant i$, $DT = \emptyset$ in $\gamma_j$.*

**Proof.** Assume that $DT \neq \emptyset$ in configuration $\gamma_0$. Let $x$ be the minimal distance from any leaf processor $q$ to a processor $p \in DT$, i.e., $x = \min\{d(p, q) ::\ p \in DT, q \in L\}$. Let $p$ and $q$ be processors in $DT$ and $L$, respectively, such that $d(p, q) = x$. No processor $p' \in C_p$ (i.e., $N_p \setminus \{P_p\}$) has $S_{p'}$ equal to $p$ (otherwise, $x$ is not minimal). So, Action $OC$ is enabled at $p$ (i.e., $p$ is either an abnormal root or $\{p, P_p\}$ forms a cycle). By Lemma 4.5, at the beginning of the next round, $\min\{d(p, q) ::\ p \in DT, q \in L\} = x + 1$. By induction, after at most $h - 1$ rounds, $DT$ is empty. Moreover, from Action $O1$, the only possible way that a processor $p$ has to choose its topological parent $P_p$ is to have exactly one predecessor $q$ such that $q \neq P_p$. So, if $DT$ is empty, there is no way for a processor to choose $P_p$ by executing Action $O1$. Thus, if $DT$ is empty, then $DT$ remains empty forever. □

It is clear that once $DT = \emptyset$ forever (Lemma 4.6), Algorithm $DFTC_U$ follows the same behavior as Algorithm $DFTC_O$. So, by Lemma 4.6, Theorem 3.2, and Remark 2.1, we can claim:

**Theorem 4.1** (*Snap-stabilizing*). *Algorithm $DFTC_U$ is a snap-stabilizing DFTC algorithm.*

### *4.3. State optimality*

As presented in Section 1, both results of [19] and [6] leads to the following theorem:

**Theorem 4.2** (*Petit and Villain [19], Cournier et al. [6]*). *To implement a snap-stabilizing DFTC cycle on an un-oriented rooted tree network, the number of states per processor p must be greater than or equal to*:

1. $\Delta_p + 2$, *if* $p \neq \mathtt{R}$ *and* $\Delta_p \geqslant 2$,
2. $\Delta_p + 1$, *otherwise.*

From Theorem 4.1 and Theorem 4.2, we can claim the following theorem:

**Theorem 4.3.** *Algorithm $DFTC_U$ is optimal in terms of the number of states per processor to implement a snap-stabilizing DFTC cycle on an un-oriented rooted tree network.*

## 5. Waiting time

As we mentioned in the introduction, the DFTC has many applications in distributed systems. As an underlying module, it can be used in two different ways. The former is to infinitely and sequentially repeat DFTC cycles, e.g., in order to maintain the mutual exclusion among the processors. In this case, a cycle can be initiated only after the preceding one is completed. In the sequel, we refer to this scheme as the *sequential DFTC* scheme. The latter consists to launch an isolated DFTC cycle, for instance to participate in a global computation. In the sequel, this second scheme will be referred to as the *isolated DFTC* scheme. We now consider the waiting time for these two schemes.

## *5.1. Sequential DFTC scheme*

We begin our discussion by making an obvious and well-known observation. In any rooted tree, oriented or not, for any sequential DFTC algorithm, each edge must be visited by the token exactly twice. So, the minimal time in terms of rounds to complete a DFTC cycle is $2(n-1)$. This leads to the following remark:

**Remark 5.1.** The delay for the root to initiate a new DFTC cycle is greater than or equal to $2n-3$ rounds—$\Omega(n)$—to implement the sequential DFTC scheme on any rooted tree.

This time corresponds to the case where a DFTC cycle is requested by the upper tasks at the root, but the root just launched a DFTC cycle, i.e., $2(n-1)$, the time for the token to visit all processors, minus 1. Note that this remark also holds even in a safe environment, i.e., with no fault.

*Sequential DFTC scheme in oriented trees*: The following theorem is proven by Theorem 3.2.

**Theorem 5.1.** *The worst-case waiting time to start the DFTC cycle using Algorithm DFTC*$_O$ *is bounded by* $2n-1$ *rounds if* $|N_{\mathtt{R}}| > 1$, $2n$ *otherwise* ($\mathtt{R}$ *has a single neighbor*)—$O(n)$ *rounds in any case—to implement the sequential DFTC scheme on any oriented tree.*

More precisely, from Theorem 5.1, the extra cost is minor since it is only 2 rounds if $|N_{\mathtt{R}}| > 1$, 3 rounds if $|N_{\mathtt{R}}| = 1$—in any case, $O(1)$ rounds. Furthermore, it can occur only during the first DFTC cycle following the fault. After the first DFTC cycle, no extra round is added between two consecutive DFTC cycles—except for the special case where the root has only one neighbor, one extra round is added (refer to the proof of Theorem 3.2).

So, we can claim the following result:

**Theorem 5.2.** *The worst-case waiting time to start the DFTC cycle using Algorithm DFTC*$_O$ *is asymptotically optimal to implement the sequential DFTC scheme on any oriented tree.*

*Sequential DFTC scheme in un-oriented trees*: Remark 5.1 remains true for any un-oriented rooted tree. We can deduce from Lemma 4.6 that the system reaches a similar configuration to the case of an oriented tree (no processor is pointing toward its parent, $\forall p :: S_p \neq P_p$) in at most $h-1$ extra rounds.

From the above discussion follows:

**Theorem 5.3.** *The worst-case waiting time to start the DFTC cycle using Algorithm DFTC*$_U$ *is bounded by* $2(n-1)+h$ *rounds if* $|N_{\mathtt{R}}| > 1$, $2n+h-1$ *rounds otherwise* ($\mathtt{R}$ *has a single neighbor*)—$O(n)$ *rounds in any case—to implement the sequential DFTC scheme on any un-oriented tree.*

Hence, from Theorem 5.3 and Remark 5.1 follows:

**Theorem 5.4.** *The worst-case waiting time to start the DFTC cycle using Algorithm DFTC*$_U$ *is asymptotically optimal to implement the sequential DFTC scheme on any un-oriented tree.*

## *5.2. Isolated DFTC scheme*

We consider that the root $\mathtt{R}$ is ready to initiate an isolated DFTC cycle, even if $\mathtt{R}$ initiated some DFTC cycles previously. For instance, this behavior can be implemented by adding an identifier to each DFTC cycle. This identifier is selected such that it is different from all identifiers of circulations in progress and known by the root. Note that if the root memory is corrupted while it chooses a new token identifier which is already circulating, then the snap-stabilization property ensures that new DFTC cycle behaves accordingly to its specification, whether the system contains corrupted processors or not. So, in the sequel, we assume that, the root is *idle* in any initial configuration. If the initial configuration is normal, the root is always ready to launch any isolated DFTC cycle. Thus, the delay is 0 rounds. In the context of stabilization, the first configuration can be abnormal, e.g., the first child chosen by $\mathtt{R}$ executing Action $O1$ can be an abnormal root using the same token identifier than the new one chosen by the root.

*Isolated DFTC scheme in oriented trees*: It is easy to deduce from the proof of Theorem 3.2 the following result:

**Theorem 5.5.** *The worst-case waiting time to start the DFTC cycle using Algorithm DFTC*$_O$ *is asymptotically optimal—equal to* 1 *round—to implement the isolated DFTC scheme on any oriented tree.*

*Isolated DFTC scheme in un-oriented trees*: From the proof of Lemma 4.6, we can claim:

**Theorem 5.6.** *The worst-case waiting time to start the DFTC cycle using Algorithm DFTC*$_U$ *is equal to* $h-1$ *rounds to implement the isolated DFTC scheme on any oriented tree.*

Fig. 5 shows an example of a configuration where the waiting time is equal to $h-1$. In this example, there is an abnormal rooted path $\overrightarrow{p}$. Process $p$ is the parent of the farthest leaf from the root. Process $q$, the neighbor of $\mathtt{R}$ such that $Next_{\mathtt{R}} = q$, belongs to $\overrightarrow{p}$.

## 6. Conclusion

We proposed two DFTC algorithms for rooted tree networks. The former works on oriented trees, the latter on un-oriented trees. Both work under any daemon, even unfair. They are time optimal (snap-stabilizing) and state optimal. Used to implement the sequential DFTC scheme, both algorithms are asymptotically optimal in term of waiting time. As an isolated DFTC, the algorithm for oriented trees has a waiting time which is

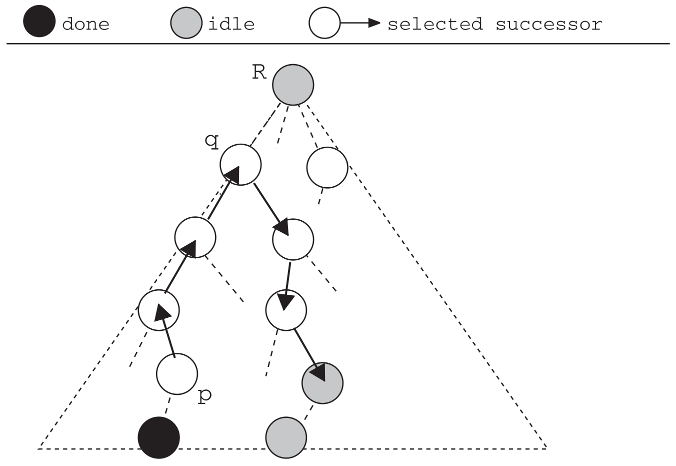

Fig. 5. An example showing the worst-case waiting time in an un-oriented tree.

asymptotically optimal. The worst-case waiting time of the second algorithm is equal to $h - 1$ rounds. We conjecture that this time is asymptotically optimal.

## References

[1] A. Arora, MG. Gouda, Distributed reset, IEEE Trans. Comput. 43 (1994) 1026–1038.
[2] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, G. Varghese, Time optimal self-stabilizing synchronization, in: STOC93 Proceedings of the 25th Annual ACM Symposium on Theory of Computing, 1993, pp. 652–661.
[3] A. Bui, AK. Datta, F. Petit, V. Villain, State-optimal snap-stabilizing PIF in tree networks, in: Proceedings of the Fourth Workshop on Self-Stabilizing Systems, IEEE Computer Society Press, Silver Spring, MD, 1999, pp. 78–85.
[4] NS. Chen, HP. Yu, ST. Huang, A self-stabilizing algorithm for constructing spanning trees, Inform. Processing Lett. 39 (1991) 147–151.
[5] Z. Collin, S. Dolev, Self-stabilizing depth-first search, Inform. Processing Lett. 49 (1994) 297–301.
[6] A. Cournier, A.K. Datta, F. Petit, V. Villain, Optimal snap-stabilizing PIF in un-oriented trees, in: OPODIS 2001, Fifth International Conference on Principles of Distributed Systems Proceedings, 2001, pp. 71–90.
[7] A. Cournier, S. Devismes, F. Petit, V. Villain, Snap-stabilizing depth-first search on arbitrary networks, in: OPODIS 2004, Eighth International Conference on Principles of Distributed Systems Proceedings, 2004, pp. 187–195.
[8] AK. Datta, C. Johnen, F. Petit, V. Villain, Self-stabilizing depth-first token circulation in arbitrary rooted networks, Distributed Comput. 13 (4) (2000) 207–218.
[9] EW. Dijkstra, Self stabilizing systems in spite of distributed control, Commun. Assoc. Comput. Mach. 17 (1974) 643–644.
[10] S. Dolev, A. Israeli, S. Moran, Self-stabilization of dynamic systems assuming only read/write atomicity, Distributed Comput. 7 (1993) 3–16.
[11] S. Dolev, A. Israeli, S. Moran, Uniform dynamic self-stabilizing leader election, IEEE Trans. Parallel Distributed Syst. 8 (4) (1997) 424–440.
[12] MG. Goudam, FF. Haddix, The stabilizing token ring in three bits, J. Parallel Distributed Comput. 35 (1996) 43–48.
[13] ST. Huang, NS. Chen, Self-stabilizing depth-first token circulation on networks, Distributed Comput. 7 (1993) 61–66.
[14] C. Johnen, G. Alari, J. Beauquier, AK. Datta, Self-stabilizing depth-first token passing on rooted networks, in: WDAG97 Distributed Algorithms 11th International Workshop Proceedings, Lecture Notes in Computer Science, vol. 1320, Springer, Berlin, 1997, pp. 260–274.
[15] C. Johnen, J. Beauquier, Space-efficient distributed self-stabilizing depth-first token circulation, in: Proceedings of the Second Workshop on Self-Stabilizing Systems, 1995, pp. 4.1–4.15.
[16] F. Petit, Highly space-efficient self-stabilizing depth-first token circulation for trees, in: OPODIS'97, First International Conference on Principles of Distributed Systems Proceedings, 1997, pp. 221–235.
[17] F. Petit, Fast self-stabilizing depth-first token circulation, in: Proceedings of the Fifth Workshop on Self-Stabilizing Systems, Lecture Notes in Computer Science, vol. 2194, Springer, Berlin, 2001, pp. 200–215.
[18] F. Petit, V. Villain, Color optimal self-stabilizing depth-first token circulation, in: I-SPAN'97, Third International Symposium on Parallel Architectures, Algorithms and Networks Proceedings, IEEE Computer Society Press, Silver Spring, MD, 1997, pp. 317–323.
[19] F. Petit, V. Villain, Optimality and self-stabilization in rooted tree networks, Parallel Processing Lett. 10 (1) (2000) 3–14.
[20] V. Villain, A new lower bound for self-stabilizing mutual exclusion algorithms, Technical Report RR97-17, LaRIA, University of Picardie Jules Verne, 1997.
[21] V. Villain, A key tool for optimality in the state model, in: Proceedings of DIMACS Workshop on Distributed Data and Structures, Carleton University Press, Oxford, 1999, pp. 133–148.

**Franck Petit** received a Ph.D. in computer science from the University of Picardie Jules Verne (France) in 1998.
He is currently a professor at the same university. His research focusses on algorithmic aspects of synchronization, stabilization and fault-tolerance in distributed systems.

**Vincent Villain** is a professor of computer science at the University of Picardie Jules Verne, Amiens, France.
He is the deputy head of the LaRIA (Laboratoire de Recherche en Informatique d'Amiens). His primary research areas are distributed computing and self-stabilization.