A Snap-Stabilizing DFS with a Lower Space Requirement*

Alain Cournier, Stéphane Devismes, and Vincent Villain

LaRIA, CNRS FRE 2733,
Université de Picardie Jules Verne, Amiens (France)
{cournier, devismes, villain}@laria.u-picardie.fr

Abstract. A *snap-stabilizing protocol*, starting from any arbitrary initial configuration, always behaves according to its specification. In [4], we presented the first snap-stabilizing depth-first search (DFS) wave protocol for arbitrary rooted networks working under an unfair daemon. However, this protocol needs $O(N^N)$ states per processors (where N is the number of processors) and needs ids on processors. In this paper, we propose an original snap-stabilizing solution for this problem with a strongly enhanced space complexity, i.e., $O(\Delta^2 \times N)$ states where Δ is the degree of the network. Furthermore, this new protocol does not need a completely identified network: only the root needs to be identified, i.e., the network is *semi-anonymous*.

1 Introduction

In an arbitrary rooted network, a Depth-First Search (DFS) Wave is initiated by the root. In this wave, all the processors are sequentially visited in depth-first search order. This scheme has many applications in distributed systems. For example, the solution of this problem can be used for solving mutual exclusion, spanning tree computation, constraint programming, routing, or synchronization.

The concept of *self-stabilization* [7] is the most general technique to design a system tolerating arbitrary transient faults. A self-stabilizing system, regardless of the initial states of the processors and messages initially in the links, is guaranteed to converge to the intended behavior in finite time. *Snap-stabilization* was introduced in [2]. A *snap-stabilizing* protocol guaranteed that it always behaves according to its specification. In other words, a snap-stabilizing protocol is also a self-stabilizing protocol which stabilizes in 0 time unit. Obviously, a *snap-stabilizing* protocol is optimal in stabilization time.

Related Works. Several self-stabilizing (but not snap-stabilizing) wave protocols based on the *depth-first token circulation* ($DFTC$) have been proposed for arbitrary rooted networks, e.g., [9,11,10,6]. All these papers have a stabilization

* A full version of this paper is available at `www.laria.u-picardie.fr/~devismes/tr2005-05.pdf`

time in $O(D \times N)$ rounds where N is the number of processors and D is the diameter of the network. The protocols proposed in [11,10,6] attempted to reduce the memory requirement from $O(\Delta \times N)$ [9] to $O(\Delta)$ states per processor where Δ is the degree of the network. However, the correctness of all the above protocols is proven assuming a (weakly) fair daemon. Roughly speaking, a daemon is considered as an adversary which tries to prevent the protocol to behave as expected, and fairness means that the daemon cannot prevent forever a processor to execute an enabled action.

The first snap-stabilizing $DFTC$ has been proposed in [12] for tree networks. In arbitrary networks, a *universal transformer* providing a snap-stabilizing version of any (neither self- nor snap-) protocol is given in [3]. Obviously, combining this protocol with any $DFTC$ protocol, we obtain a snap-stabilizing $DFTC$ protocol for arbitrary networks. However, the resulting protocol works assuming a weakly fair daemon only. Indeed, it generates an infinite number of snapshots, independently of the token progress. Therefore, the number of steps per wave cannot be bounded. Finally, we propose in [4] the first snap-stabilizing DFS protocol for arbitrary rooted network assuming an unfair daemon, i.e., the weakest scheduling assumption. In contrast with the previous solutions, the time complexity of each wave of the protocol can now be bounded in terms of steps.

Contribution. The protocol of [4] works on identified networks and needs $O(N^N)$ states per processor. In this paper, we reduce this space complexity to $O(\Delta^2 \times N)$ states per processor using a method similar to that in [1]. This new solution also works assuming an unfair daemon. Moreover, our protocol does not need a completely identified network: only the root needs to be identified, i.e., the network is *semi-anonymous.* Unfortunately, the time complexities of our protocol are greater than those in [4]: a complete DFS Wave needs $O(N^2)$ rounds and $O(N^3)$ steps instead of $O(N)$ rounds and $O(N^2)$ steps. Nevertheless, the gain of space requirement is such that the worst time complexities are a minor drawback.

Outline of the Paper. The rest of the paper is organized as follows: in Section 2, we describe the model in which our protocol is written. Moreover, in the same section, we give a formal statement of the Depth-First Search Wave Protocol solved in this paper. In Section 3, we present the protocol and the intuitive ideas of its correctness (due to the lack of space, the proof of correctness has been omitted). Finally, we make concluding remarks in Section 4.

2 Preliminaries

Network. We consider a *network* as an undirected connected graph $G = (V, E)$ where V is a set of *processors* ($|V| = N$) and E is the set of *bidirectional communication links.* We consider networks which are *asynchronous* and *rooted,* i.e., among the processors, we distinguish a particular processor called *root.* We denote the root processor by r. A communication link (p, q) exists if and only if p and q are neighbors. Every processor p can distinguish all its links. To simplify the presentation, we refer to a link (p, q) of a processor p by the *label* q. We assume that the labels of p, stored in the set $Neig_p$, are locally ordered by $\prec_p$. We assume that $Neig_p$ is a constant and shown as an input from the system.

Computational Model. In our model, each processor executes the same program except r. We consider the local shared memory model of computation. The program of every processor consists in a set of *shared variables* (henceforth, referred to as variables) and a finite set of *actions*. A processor can write to its own variable only, and read its own variables and that of its neighbors. Each action is constituted as follows: $<$ *label* $>$ :: $<$ *guard* $>$ $\rightarrow$ $<$ *statement* $>$. The guard of an action in the program of p is a boolean expression involving variables of p and its neighbors. The statement of an action of p updates one or more variables of p. An action can be executed only if its guard is satisfied. We assume that the actions are atomically executed, i.e., the evaluation of a guard and the execution of the corresponding statement, if executed, are done in one atomic step.

The *state* of a processor is defined by the value of its variables. The *state* of a system is the product of the states of all processors. We will refer to the state of a processor and the system as a (*local*) *state* and (*global*) *configuration*, respectively. We note $\mathcal{C}$ the set of all possible configuration of the system. Let $\gamma \in \mathcal{C}$ and A an action of p ($p \in V$). A is said *enabled* in γ if the guard of A is satisfied in γ. Processor p is said to be *enabled* in γ if it has an enabled action in γ.

Let a distributed protocol $\mathcal{P}$ be a collection of binary transition relations denoted by $\mapsto$, on $\mathcal{C}$. 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 $i \geq 0$, $\gamma_i \mapsto \gamma_{i+1}$ (called a *step*) if γ_{i+1} exists, else γ_i is a terminal configuration. *Maximality* means that the sequence is either finite (and no action of $\mathcal{P}$ is enabled in the terminal configuration) or infinite. All computations considered in this paper are assumed to be maximal. The set of all possible computations of $\mathcal{P}$ is denoted by $\mathcal{E}$.

As we have already said, each execution is decomposed into steps. Each step is shared into three sequential phases atomically executed: (*i*) every processor evaluates its guard, (*ii*) a *daemon* (also called *scheduler*) chooses some enabled processors, and (*ii*) the chosen processors execute some of their enabled actions. When these three phases are done, the next step begins.

A *daemon* can be defined in terms of *fairness* and *distributivity*. In this paper, we use the notion of *weakly fairness*: if a daemon is *weakly fair*, then every continuously enabled processor is eventually chosen (by the daemon) to execute an action. We also use the notion of *unfairness*: the *unfair* daemon can forever prevent a processor to execute an action except if it is the only enabled processor. Concerning the *distributivity*, we assume that the daemon is *distributed* meaning that, at each step, if one or more processor are enabled, then the daemon chooses at least one (possibly more) of these processors to execute actions.

We consider that any processor p executed a *disabling action* in the computation step $\gamma_i \mapsto \gamma_{i+1}$ if p was *enabled* in γ_i and not enabled in γ_{i+1}, but did not execute any action between these two configurations. (The disabling action represents the following situation: at least one neighbor of p changes its state between γ_i and γ_{i+1}, and this change effectively made the guard of all actions of p false.)

To compute the time complexity, we use the definition of *round* [8]. This definition captures the execution rate of the slowest processor in any computation. Given a computation e ($e \in \mathcal{E}$), the *first round* of e (let us call it e') is the minimal prefix of e containing the execution of one action (an action of the protocol or the disabling action) of every enabled processor from the first configuration. Let e'' be the suffix of e such that $e = e'e''$. The *second round* of e is the first round of e'', and so on.

In order to make our protocol more readable, we design it as a *composition* of four algorithms. In this composition, if a processor p is enabled for k of the combined algorithms, then, if the daemon chooses it, p executes an enabled action of each of the k algorithms, in the same step. Variables, predicates, or macros of Algorithm A used by Algorithm B are shown as inputs in Algorithm B.

Snap-Stabilizing Systems. Snap-stabilization [2] is a general concept which can be apply to several kinds of distributed protocol. However, the protocol presented in this paper is a *wave protocol* as defined by Tel in [13]. So, we now propose a simpler definition of snap-stabilization holding for wave protocols:

Definition 1. *[Snap-stabilization for Wave Protocols] Let $\mathcal{T}$ be a task, and $\mathcal{SP}_\mathcal{T}$ a specification of $\mathcal{T}$. A wave protocol $\mathcal{P}$ is snap-stabilizing for $\mathcal{SP}_\mathcal{T}$ if and only if: (i) at least one processor eventually executes a particular action of $\mathcal{P}$, and (ii) the result obtained with $\mathcal{P}$ from this particular action always satisfies $\mathcal{SP}_\mathcal{T}$.*

Specification of the Depth-First Search Wave Protocol.

Specification 1. *Let* Visited *be a set of processors. A finite computation $e \in \mathcal{E}$ is called a DFS Wave if and only if: (i)* r *initiates the DFS Wave by initializing* Visited *with r, (ii) all other processors are then sequentially included in* Visited *in DFS order, and (iii)* r *eventually detects the termination of the process.*

Remark 1. So, in the practice, to prove that our protocol is snap-stabilizing we have to show that every execution of the protocol satisfies: (i) r eventually initiates a DFS Wave, and (ii) thereafter, the execution satisfies Specification 1.

3 Algorithm

We now present an informal description of our DFS Wave protocol (see Algorithms 1 to 8 for the formal description). Along this description, we will give the main keys to understand why our protocol is snap-stabilizing. For a sake of clarity, we divided our protocol, referred to as Algorithm $\mathcal{DFS}$, into four phases:

1. The *visiting phase* (Algorithms 1 and 2) sequentially visits the processors in depth-first search order: Starting from r, the visit progresses as deeply as possible in the network. When the visit cannot progress anymore (i.e., the

visit reaches a processor with a completely visited neighbourhood), the visit backtracks to the latest visited processor having some non-visited neighbors, if any. The visit terminates when it backtracks to r and r has a completely visited neighbourhood.

2. The *cleaning phase* (Algorithms 3 and 4) cleans the trace of the last visiting phase so that the root is eventually ready to initiate another visiting phase again. The cleaning phase is initiated only when the visiting phase is entirely done.
3. The *confirmation phase* (Algorithms 5 and 6) prevents to forgot some processors in the visiting phase initiated by the root (especially when the system contains some erroneous behaviors). The confirmation phase is performed each time the protocol needs to be sure that all the neighbors of the latest visited processor have been visited by the *normal* visiting phase, i.e., the visiting phase from the root. Indeed, since the system starts from any configuration, some visiting phases can be rooted at another processor than r, i.e., the *abnormal* visiting phases.
4. The *abnormal trees' deletion* (Algorithms 7 and 8) erases all the *abnormal* visiting phases.

In order to more precisely describe this four phases, we first present how to implement a non self-stabilizing DFS protocol (visiting phase and cleaning phase). We then explain the problems appearing when we use this protocol in a self-stabilizing[1] context and how to solve them (abnormal trees' deletion, confirmation phase, ...). In particular, we will present some tools used for insuring the snap-stabilization of our protocol, i.e., the optimality of our protocol in terms of stabilization time.

Algorithm 1. *Visiting Phase* for $p = r$

```
Inputs:
 on read: $Neig_p$: set of neighbors (locally ordered);
          $Child_p$: macro of the abnormal trees' deletion;
 on read/write: $Que_p$: variable of the confirmation phase;
Constants:   $L_p = 0$; $Par_p = \perp$;
Variable:    $S_p \in Neig_p \cup \{idle, rdone\}$;
Macro:
 $Next_p$      $= (q = \min_{\prec_p}\{q' \in Neig_p :: S_{q'} = idle\})$ if $q$ exists, $rdone$ otherwise;
 $RealChild_p$ $= \{q \in Child_p :: E_q \neq E_p \Rightarrow E_p = B\}$;   /* valid for $S_p \neq idle$ only */
Predicates:
 $End(p)$        $\equiv (\forall q \in Neig_p :: S_q \neq idle \Rightarrow Par_q \neq p)$
 $AnswerOK(p)$   $\equiv (Que_p = A) \wedge (\forall q \in Neig_p :: S_q \neq idle \Rightarrow Que_q = A)$
 $Forward(p)$    $\equiv (S_p = idle) \wedge (\forall q \in Neig_p :: S_p = idle)$
 $Backward(p)$   $\equiv (\exists q \in Neig_p :: S_p = q \wedge Par_q = p \wedge S_q = done)$
                 $\wedge\ [AnswerOK(p) \vee (\exists q \in Neig_p :: S_q = idle)]$
Actions:
 $F\text{-}action :: Forward(p) \rightarrow S_p := Next_p;\ Que_p := Q$;
 $B\text{-}action :: Backward(p) \rightarrow S_p := Next_p$;
```

[1] Remember that our snap-stabilizing protocol is a self-stabilizing protocol which stabilizes in 0 time unit.

Visiting Phase. In our non self-stabilizing protocol, each processor p maintains two variables to implement this phase[2]:

- $\forall p \in V$, $S_p \in Neig_p \cup \{idle, done\}$ if $p \neq r$ and $S_p \in Neig_p \cup \{idle, rdone\}$ if $p = r$. $S_p = idle$ means that p is ready to be visited. $S_p = q$ such that $q \in Neig_p$ means that p participates to a visiting phase and its *successor* in the visit is its neighbor q (respectively, p is called the *predecessor* of q). Finally, S_p is set to *done* (resp. *rdone* if $p = r$) when the visit locally terminates at p.
- $\forall p \in V$, Par_p is used for keeping a mark of the DFS spanning tree computed by the protocol. Indeed, Par_p designates the *parent* of p in the traversal: when p is visited for the first time, it designates its predecessor with Par_p. Obviously, r never has any parent. So, we state that Par_r is the constant $\perp$.

Since our protocol is non self-stabilizing, any execution must start from particular configurations. We call these configurations the *normal initial configurations.* Here, there is only one normal initial configuration for our protocol and it is defined as follows: $\forall p \in V$, $S_p = idle$. In this configuration, the root r initiates a visiting phase by pointing out using S_r to its minimal neighbor p in the local order $\prec_r$ (*F-action*). By this action, r becomes the only *visited* processor. Then, at each step, exactly one processor p is enabled and two cases are possible:

a) $S_p = idle$ and $\exists q \in Neig_p$ such that $S_q = p$. In this case, $p \neq r$ and p executes *F-action* to be visited for the first time. First, p points out to q (its predecessor) using Par_p. Then, p computes S_p as follows:
   - **If** p has still some non-visited neighbors, i.e., $\exists p' \in Neig_p$ such that $S_{p'} = idle$, then p chooses its minimal non-visited neighbor by $\prec_p$ as its successor in the traversal.
   - **Otherwise** p sets S_p to *done*.

b) $S_p = q$ ($q \in Neig_p$) and $S_q = done$, i.e., the visiting phase backtracks to p because the visit from q is terminated. In this case p executes *B-action*:
   - **If** p has still some non-visited neighbors, then it updates S_p by pointing out to a new successor (its minimal non-visited neighbor by $\prec_p$).
   - **Otherwise** it sets S_p to *done* (resp. *rdone* if $p = r$).

Therefore, step by step, the visiting phase dynamically built a spanning tree of the network rooted at r (w.r.t. the Par variable), noted $Tree(r)$. It is easy to see that this phase follows a DFS order and the number of steps required for the phase is $2N - 1$. Moreover, since the behavior is sequential, the number of rounds is the same. Finally, S_r is eventually set to *rdone* meaning that the visiting phase is terminated for all processors. By this latter action, r initiates the cleaning phase.

Cleaning Phase. The aim of the cleaning phase is to erase the trace of the last visiting phase in order to bring the system in the normal initial configuration

[2] This phase does not exactly correspond to Algorithms 1 and 2. Indeed, we will see later that this phase must be modified in order to run correctly in a self-stabilizing context.

Algorithm 2. *Visiting Phase* for $p \neq r$

```
Inputs:
 on read: $Neig_p$: set of neighbors (locally ordered);
          $Child_p$: macro of the abnormal trees' deletion;
 on read/write: $Que_p$: variable of the confirmation phase;
                $E_p$: variable of the abnormal trees' deletion;
Variables:  $S_p \in Neig_p \cup \{idle, wait, done, rdone\}$; $Par_p \in Neig_p$; $L_p \in \mathbb{N}$;
Macros:
 $WaitOrDone_p$ = $wait$ if $(S_p = idle)$, $done$ otherwise;
 $Next_p$       = $(q = \min_{\prec_p}\{q' \in Neig_p :: S_{q'} = idle\})$ if $q$ exists, $WaitOrDone_p$ otherwise;
 $Pred_p$       = $\{q \in Neig_p :: S_q = p \wedge E_q = C\}$;
 $RealChild_p$  = $\{q \in Child_p :: E_q \neq E_p \Rightarrow E_p = B\}$;   /* valid for $S_p \neq idle$ only */
Predicates:
 $End(p)$        $\equiv (\forall q \in Neig_p :: S_q \neq idle \Rightarrow Par_q \neq p)$
 $AnswerOK(p)$   $\equiv (Que_p = A) \wedge (\forall q \in Neig_p :: S_q \neq idle \Rightarrow Que_q = A)$
 $Forward(p)$    $\equiv (S_p = idle) \wedge (|Pred_p| = 1) \wedge End(p)$
 $WaitOk(p)$     $\equiv Normal(p) \wedge (S_p = Wait) \wedge [AnswerOK(p) \vee (\exists q \in Neig_p :: S_q = idle)]$
 $Backward(p)$   $\equiv Normal(p) \wedge (\exists q \in Neig_p :: S_p = q \wedge Par_q = p \wedge S_q = done)$
                 $\wedge [AnswerOK(p) \vee (\exists q \in Neig_p :: S_q = idle)]$
 $BadSucc(p)$    $\equiv Normal(p) \wedge AnswerOk(p)$
                 $\wedge (\exists q \in Neig_p :: S_p = q \wedge q \notin RealChild_p \wedge S_q \neq idle)$
Actions:
 F-action    :: $Forward(p)$  → $Par_p := (q \in Pred_p)$; $S_p := Next_p$;
                                $Que_p := Q$; $L_p := L_{Par_p} + 1$; $E_p := C$;
 Fbis-action :: $WaitOk(p)$   → $S_p := Next_p$;
 B-action    :: $Backward(p)$ → $S_p := Next_p$;
 IE-action   :: $BadSucc(p)$  → $S_p := Next_p$;
```

Algorithm 3. *Cleaning Phase* for $p = r$

```
Inputs:
 on read: $Neig_p$: set of neighbors (locally ordered);
          $End(p)$: predicate of the visiting phase;
 on read/write: $S_p$: variable of the visiting phase;
Predicate:
 $Clean(p) \equiv (S_p = rdone) \wedge End(p) \wedge (\forall q \in Neig_p :: S_q \in \{idle, rdone\})$
Action:
 C-action :: $Clean(p)$ → $S_p := idle$;
```

again ($\forall p \in V$, $S_p = idle$). Only the root can detect if the visiting phase is entirely done: when r sets S_r to $rdone$. Since the root detects the end of the visiting phase, the $rdone$ value is propagated toward the leaves of $Tree(r)$ following the Par variables (*RD-action*) to inform all processors of this termination (to that goal, we add the state $rdone$ into the definition of S_p, $\forall p \in V \setminus \{r\}$). Then, each leaf of $Tree(r)$ successively cleans itself by setting its S variable to $idle$ (*C-action*). Therefore, the system eventually reaches the normal initial configuration again. This phase adds $2N - 1$ steps and the number of additional rounds is $2H + 1$ where H is the height of the tree computed during the visiting phase (n.b., H is bounded by $N - 1$).

We presented a non self-stabilizing DFS protocol. Of course, in the context of self-stabilization, this protocol does not work correctly. So, we must modify its existing actions as well as we must add some other actions. In particular, we introduce the *confirmation phase* to guarantee that a visiting phase initiated by the root eventually visits **all** processors. This latter point is crucial to obtain a snap-stabilizing protocol.

Algorithm 4. *Cleaning Phase* for $p \neq r$

```
Inputs:
 on read: $Neig_p$: set of neighbors (locally ordered);
          $Par_p$: variable of the visiting phase;
          $End(p)$: predicate of the visiting phase;
          $Normal(p)$: predicate of the abnormal trees' deletion;
 on read/write: $S_p$: variable of the visiting phase;
Predicates:
 $RdonePar(p) \equiv Normal(p) \wedge (S_p = done) \wedge (S_{Par_p} = rdone)$
 $Clean(p) \equiv Normal(p) \wedge (S_p = rdone) \wedge End(p) \wedge (\forall q \in Neig_p :: S_q \in \{idle, rdone\})$
Actions:
 $RD\text{-}action :: RdonePar(p) \rightarrow S_p := rdone;$
 $C\text{-}action :: Clean(p) \rightarrow S_p := idle;$
```

Modifying the Existing Actions. Starting from any arbitrary configuration, the system may contain successors' cycles. We detect this error by using a new variable L for each processor. For the root, L_r is the constant 0. Each other processor p dynamically computes its L variable each time it executes *F-action*: $L_p := L_q + 1$ where q is the predecessor of p (n.b., p does not execute *F-action* if it has several predecessors). Typically, L_p contains the length of the path from the root r to p w.r.t. the variable Par. Obviously, in a cycle of successors, at least one processor p satisfies $L_p \neq L_{Par_p} + 1$.

In the visiting phase, we have shown that a visited processor p sets S_p to *done* (resp. *rdone* if $p = r$) when all its neighbors have been visited. In the non self-stabilizing context, p can easily detect when all neighbors are visited: when $\forall p' \in Neig_p$, $S_{p'} \neq idle$. However, in a self-stabilizing scheme, since some neighbors of p may belong to abnormal visiting phases, this condition is not sufficient. So, in order to guarantee that every neighbor of p is visited, we introduce a new phase, the *confirmation phase*. The aim of this phase is to insure that a processor p, participating to a visiting phase initiated by r, sets S_p to *done* (resp. *rdone* if $p = r$) only when its neighbourhood is completely visited by the visiting phase from r. To apply this concept, we add the state *wait* into the definition of S_p, $\forall p \in V \setminus \{r\}$ and we change *F-action* of the initial protocol: when a processor $p \neq r$ receives a visiting phase (*F-action*) and satisfies $\forall p' \in Neig_p$, $S_{p'} \neq idle$, it now sets S_p to *wait* instead of *done* (see Macros $Next_p$ and $WaitOrDone_p$). This action initiates the *confirmation phase.* We now describe in details the *confirmation phase.*

Confirmation Phase. To implement this phase, we introduce a new variable Que_p for each processor p: $Que_p \in \{Q, R, W, A\}$ if $p \neq r$ and $Que_p \in \{Q, R, A\}$ if $p = r$. The Q and R value are used for resetting the part of the network which is concerned by the confirmation phase. The W value corresponds to the request of a processor: "Have I terminated my visiting phase?". The A value corresponds to the answer sending by the root (n.b., the root is the only processor able to generate a A value). We now explain how this phase works.

The confirmation phase concerns processors such that $S \neq idle$ only. Variable Que_p of a processor p is initialized by *F-action* of the visiting phase: $Que_p := Q$. This value forces all its neighbors satisfying $S \neq idle$ to execute $Que := R$ (*R-action*). When all the neighbors of p have reset, p also executes $Que_p := R$. Then, the R values are propagated up as far as possible following the Par variable (*R-action*). By this mechanism, the A values are deleted in the Par paths of p and its neighbors (in particular, the A values present since the initial configuration). Thus, from now on, when a A value reaches a requesting processor, this value cannot come from anyone but r and the processor obviously belongs to the normal visiting phase. Now, as we have seen before, a processor q ($q \neq r$) eventually waits a confirmation from the root that all its neighbors are visited ($S_q = wait$). In this case, q and its neighbors satisfying $S = done$ execute *W-action*[3], i.e., $Que := W$ meaning that they are now waiting for an answer from r. W value is propagated toward the root if possible (a processor p propagates the W value when all its children satisfies $Que \notin \{Q, R\}$). When the W value reaches the children of r, r executes its *A-action*: it sends an answer (A) to its children and so on. So, if q and its neighbors receive this answer (A), q is sure that it and all its neighbors belong to $Tree(r)$. In this case, q satisfies $AnswerOk(q)$ and executes *Fbis-action* to set S_q to *done* meaning that the visiting phase from it is now terminated and so on.

We now explain why a visited processor needs to initiate the *confirmation phase* only once: when it executes *F-action*. Assume that a processor of the visiting phase initiated by r, p, has a neighbor q such that q belongs to an abnormal visiting phase. We have already seen that, when p is visited for the first time (*F-action*), it also executes $Que_p := Q$. This action initiates the *confirmation phase* and forces q to execute $Que_q := R$ (*R-action*). This value erases all A values in the Par path of q. Since only r can generate a A value, q never receives any A. Thus, p cannot satisfy $AnswerOk(p)$ while q belongs to an abnormal visiting phase. As a consequence, p cannot set S_p to *done* (resp. *rdone* if $p = r$) while one of its neighbor belongs to an abnormal visiting phase (see *Fbis-action* and *B-action*).

Unfortunately, starting from the normal initial configuration, this phase strongly slows down the initial protocol, nevertheless, it does not generate any deadlock. The worst case (starting from the normal initial configuration) is obtained with a computed tree of height in $O(N)$ with a number of leaves also in $O(N)$ at a distance in $O(N)$ of the root. In that case, $\Theta(N)$ *W-actions* are initiated by these leaves. So, the complexity is in $\Theta(N^2)$ steps. Due to the sequentiality along the path from a requesting leaf to the root, the complexity in terms of rounds is also in $\Theta(N^2)$.

Of course, we have now to deal with abnormal configurations. We first introduce a new action in the visiting phase to remove some deadlock due to the variables' initial configurations of the visiting phase itself.

IE-Action. To prevent the system from any deadlock, we add *IE-action* to solve a case which can only appear in the initial configuration: The active end of $Tree(r)$ (i.e., the only processor $p \in Tree(r)$ such that $End(p) \wedge S_p \notin \{done,$

[3] Starting from an arbitrary configuration, the neighbors of q satisfying $S = done$ can belong to $Tree(r)$, so, they must also receive an acknowledgment from r so that q knows that they are not in an abnormal visiting phase.

Algorithm 5. *Confirmation Phase* for $p = r$

```
Inputs:
 on read: $Neig_p$: set of neighbors (locally ordered);
          $S_p$: variable of the visiting phase;
          $RealChild_p$: predicate of the visiting phase;
Variable:   $Que_p \in \{Q, R, A\}$;
Predicates:
 $Require(p) \equiv (S_p \neq idle) \wedge [[Que_p = Q \wedge (\forall q \in Neig_p :: S_q \neq idle \Rightarrow Que_q \in \{Q, R\})]$
             $\vee [Que_p = A \wedge (\exists q \in Neig_p :: (S_q \neq idle \wedge Que_q = Q)$
             $\vee (q \in RealChild_p \wedge Que_q = R))]]$
 $Answer(p) \equiv (S_p \neq idle) \wedge (Que_p = R) \wedge (\forall q \in RealChild_p :: Que_q \in \{W, A\})$
             $\wedge (\forall q \in Neig_p :: S_q \neq idle \Rightarrow Que_q \neq Q)$
Actions:
 R-action :: $Require(p) \rightarrow Que_p := R$;
 A-action :: $Answer(p) \rightarrow Que_p := A$;
```

Algorithm 6. *Confirmation Phase* for $p \neq r$

```
Inputs:
 on read: $Neig_p$: set of neighbors (locally ordered);
          $S_p$, $Par_p$: variables of the visiting phase;
          $RealChild_p$: macro of the visiting phase;
          $End(p)$: predicate of the visiting phase;
          $Normal(p)$: predicate of the abnormal trees' deletion;
Variable:   $Que_p \in \{Q, R, W, A\}$;
Predicates:
 $Require(p)$      $\equiv Normal(p) \wedge (S_p \neq idle)$
                   $\wedge [[Que_p = Q \wedge (\forall q \in Neig_p :: S_q \neq idle \Rightarrow Que_q \in \{Q,R\})]$
                   $\vee [Que_p \in \{W,A\} \wedge (\exists q \in Neig_p :: (S_q \neq idle \wedge Que_q = Q)$
                   $\vee (q \in RealChild_p \wedge Que_q = R))]]$
 $WaitAnswer(p)$   $\equiv Normal(p) \wedge (S_p \neq idle) \wedge (Que_p = R) \wedge (Que_{Par_p} = R)$
                   $\wedge (\forall q \in Neig_p :: S_q \neq idle \Rightarrow Que_q \neq Q)$
                   $\wedge [End(p) \Rightarrow (S_p \neq rdone \wedge (S_p \in Neig_p \Rightarrow S_{S_p} \neq idle))]$
                   $\wedge [\neg End(p) \Rightarrow (\forall q \in RealChild_p :: Que_q \in \{W,A\})]$
 $Answer(p)$       $\equiv Normal(p) \wedge (S_p \neq idle) \wedge (Que_p = W) \wedge (Que_{Par_p} = A)$
                   $\wedge (\forall q \in RealChild_p :: Que_q \in \{W, A\})$
                   $\wedge (\forall q \in Neig_p :: S_q \neq idle \Rightarrow Que_q \neq Q)$
Actions:
 R-action :: $Require(p)$    $\rightarrow Que_p := R$;
 W-action :: $WaitAnswer(p)$ $\rightarrow Que_p := W$;
 A-action :: $Answer(p)$     $\rightarrow Que_p := A$;
```

rdone}) can designate as successor a processor q (i.e., $S_p = q$) such that q belongs to $Tree(r)$. Now, thanks to the confirmation phase, p eventually knows that it does not designate a "good" successor because p and q receives an acknowledgment (A) from r. So, p eventually changes its successor by *IE-action*. In the following, *IE-action* is considered as an action of the visiting phase. This action is enough to break the deadlock of the visiting phase rooted at r. This action (and the confirmation phase associated) does not add a significant cost.

We need now to deal with abnormal visiting phases, i.e., visiting phases rooted at another processor than r. The *abnormal trees' deletion* we now introduce erases these abnormal visiting phases.

Abnormal Trees' Deletion. We first explain how to detect the abnormal visiting phases. In a normal visiting phase, each non-root processor p must maintain some properties based on the value of its variables and that of its parent. We list these conditions below:

1. If p is involved in the DFS Wave ($S_p \neq idle$) and p is designated as successor by its parent ($S_{Par_p} = p$) then S_p must be different of the *rdone* value. Indeed, the *rdone* value is generated by the root only and is propagated down in the spanning tree.
2. If p is involved in the DFS Wave and p is not designated as successor by its parent ($S_{Par_p} \neq p$) then the visiting phase from p is terminated, i.e., $S_p \in \{done, rdone\}$.
   - If $S_p = rdone$ then its parent, Par_p, must satisfy $S_{Par_p} = rdone$. Indeed, the *rdone* value is propagated from the root to the leaves of $Tree(r)$ after the end of the visiting phase.
   - If $S_p = done$ then, as $S_{Par_p} \neq p$, the visiting phase has backtracked to Par_p (by *B-action*). So, either Par_p points out to another successor, i.e., $S_{Par_p} \in Neig_{Par_p} \setminus \{p\}$; or the visiting phase from Par_p is also terminated, i.e., $S_{Par_p} \in \{done, rdone\}$. More simply, if $S_{Par_p} \neq p$ and $S_p = done$ then, $S_{Par_p} \notin \{idle, wait\}$.
3. Finally, if p is involved in the DFS Wave and p satisfies 1. and 2. (Predicate *GoodPar*(p)) then its level L_p must be equal to one plus the level of its parent (Predicate *GoodLevel*(p)).

If one of these conditions is not satisfied by p then *AbRoot*(p) is *true*. Now, starting from any configuration, p may satisfy *AbRoot*. We can then remark that the abnormal visiting phase from p shapes an abnormal tree noted $Tree(p)$: $\forall q \in V$, $q \in Tree(p)$ if and only if there exists a sequence of nodes ($p_0 = p$), ..., p_i, ..., p_k such that, $\forall i \in [1...k]$, $p_i \in Child_{p_{i-1}}$ (among the neighbors designating p_{i-1} with Par, only those satisfying $S \neq idle \wedge \neg AbRoot$ are considered as p_{i-1} children).

We now explain how the protocol cleans these abnormal trees. In order to clean the abnormal tree $Tree(p)$, we cannot simply set S_p to *idle*. Since some processors in the visiting phase can be in $Tree(p)$. If we simply set S_p to *idle*, then p can participate again to the visiting phase of the tree of which it was the root. As we do not assume the knowledge of any bound on the L values (we may assume that the maximum value of L is any upper bound of N), this scheme can progress infinitely often, and the system contains an abnormal tree which can prevent the progression of the tree of the normal visiting phase ($Tree(r)$). We solve this problem by paralyzing the progress of any abnormal tree before removing it. First, a processor p can be visited from its neighbor q only if q satisfies $S_q = p$ and $E_q = C$ (see $Pred_p$). Then, if p hooks on to $Tree(q)$ (*F-action*), it also executes $E_p := C$. If q is an abnormal root, then it sets its variable E_q to B and broadcasts this value in its tree (and only in its tree). When q receives an acknowledgment of all its children (Value F of Variable E), it knows that all the processors p of its tree have $E_p = F$ and no processor can now participate in the visiting phase from any p. So, q can leave its tree (*EC-action*) and it will be no more visited by this abnormal visit. Thus, by this mechanism, all the abnormal trees eventually disappear.

The management of the E variables adds new kind of errors. Indeed, $\forall p$, $q \in V$ such that $S_p \neq idle \wedge S_q \neq idle \wedge Par_q = p \wedge \neg AbRoot(q)$, p and

Algorithm 7. *Abnormal trees' Deletion* for $p = r$

```
Inputs:
 on read: $Neig_p$: set of neighbors (locally ordered);
          $S_p$, $Par_p$, $L_p$: variables of the visiting phase;
 Constant:    $E_p = C$;
Macros:
 $Child_p = \{q \in Neig_p :: Par_q = p \wedge S_q \neq idle \wedge L_q = L_p + 1$      /* valid for $S_p \neq idle$ only */
          $\wedge\ (S_p = q \Rightarrow S_q \neq rdone) \wedge [[S_p \neq q] \Rightarrow [(S_q = rdone \wedge S_p = rdone) \vee (S_q = done)]]\}$;
```

q must satisfy $(E_p,\ E_q) \in \{(B,C),\ (B,B),\ (B,F),\ (F,F),\ (C,C)\}$. Predicates $FCorrection(p)$, $BCorrection(p)$, and $CCorrection(p)$ allows to detect if this condition is not satisfied by p and q. Now, we can remark that these kinds of error are local and can only appear in the initial configuration. So, we simply correct it by executing $S_q := idle$ and $E_q := C$ (*EC-action*).

To remove an abnormal tree, any processor in the tree has at most three actions to execute. So, the additional cost of this phase is in $\Theta(N)$ by tree for both steps and rounds. So, in the worst case ($\Theta(N)$ abnormal trees), the cost is in $\Theta(N^2)$ steps but is still in $\Theta(N)$ rounds because trees are removed in parallel.

Nevertheless, the presence of abnormal trees in the system involves an overcost in terms of steps for the visiting, cleaning, and confirmation phases, respectively (the overcost in terms of rounds is not significant because the abnormal trees are removed in $\Theta(N)$ rounds). Indeed, each time a processor p initiates a question (in the confirmation phase), this question can be propagated to (in the worst case) all the processors of the network. So, the overcost is $O(N)$ steps for each processor ($O(N)$) of each tree ($O(N)$), i.e., $O(N^3)$ steps. Concerning now the visiting phase, a processor p such that $p \neq r$ can execute an action of the visiting phase while it is in an abnormal tree (because of the initial configuration) or can hook on to abnormal tree by *F-action*. But, in both cases, a confirmation phase will be initiated (by p or one of its descendants) before p executes another action of the visiting phase. As explained before, this phase will lock the visiting phase for p until it leaves its tree. In the same way, p may execute its cleaning phase once (*RD-action* and *C-action*) to leave its tree but the next time it hook on to an abnormal tree, it will be lock by the confirmation phase and will execute no action of the cleaning phase until it leaves the tree by the abnormal trees' deletion. So, the overcost is $O(1)$ steps for each processor ($O(N)$) of each tree ($O(N)$), i.e., $O(N^2)$ steps. Hence, globally, the presence of abnormal trees in the system involves an overcost in terms of steps which is significant for the confirmation phase only: $O(N^3)$ steps.

Snap-stabilization of the Protocol. From the previous discussion, we know that, from normal configurations (i.e., configurations containing no abnormal trees), a traversal rooted at r is completely performed (i.e., visiting, cleaning, and confirmation phases) in $O(N^2)$ steps. Also, we know that the presence of abnormal trees in the system involves an overcost of $O(N^3)$ steps (mainly due to the confirmation phase). Finally, these abnormal trees are removed from the systems in $O(N^2)$ actions of the abnormal trees' deletion. So, despite the daemon (weakly

Algorithm 8. *Abnormal trees' Deletion* for $p \neq r$

```
Inputs:
 on read: $Neig_p$: set of neighbors (locally ordered);
          $Par_p$, $L_p$: variables of the visiting phase;
 on read/write: $S_p$: variable of the visiting phase;
Variable:    $E_p \in \{B, F, C\}$;
Macros:
 $Child_p = \{q \in Neig_p :: Par_q = p \wedge S_q \neq idle \wedge L_q = L_p + 1$        /* valid for $S_p \neq idle$ only */
          $\wedge\ (S_p = q \Rightarrow S_q \neq rdone)$
          $\wedge\ [[S_p \neq q] \Rightarrow [(S_q = rdone \wedge S_p = rdone) \vee (S_q = done \wedge S_p \notin \{idle, wait\})]]\}$;
Predicates:
 $GoodLevel(p)$     $\equiv (S_p \neq idle) \Rightarrow (L_p = L_{Par_p} + 1)$
 $GoodPar(p)$       $\equiv (S_p \neq idle) \Rightarrow [[S_{Par_p} = p \Rightarrow S_p \neq rdone]$
                      $\wedge\ [(S_{Par_p} \neq p) \Rightarrow ((S_p = rdone \wedge S_{Par_p} = rdone)$
                      $\vee\ (S_p = done \wedge S_{Par_p} \notin \{idle, wait\}))]]$
 $AbRoot(p)$        $\equiv GoodPar(p) \Rightarrow \neg GoodLevel(p)$
 $FreeError(p)$     $\equiv (S_p \neq idle) \Rightarrow (E_p = C)$
 $BadC(p)$          $\equiv (S_p \neq idle \wedge E_p = C \wedge E_{Par_p} = F)$
 $Normal(p)$        $\equiv \neg AbRoot(p) \wedge FreeError(p) \wedge \neg BadC(p)$
 $BError(p)$        $\equiv (S_p \neq idle) \wedge (E_p = C) \wedge [\neg AbRoot(p) \Rightarrow (E_{Par_p} = B)]$
                      $\wedge\ (\forall q \in Child_p :: E_q = C)$
 $FError(p)$        $\equiv (S_p \neq idle) \wedge (E_p = B) \wedge [\neg AbRoot(p) \Rightarrow (E_{Par_p} = B)]$
                      $\wedge\ (\forall q \in Child_p :: E_q = F)$
 $FAbRoot(p)$       $\equiv (E_p = F) \wedge AbRoot(p)$
 $BCorrection(p)$   $\equiv (S_p \neq idle) \wedge (E_p = B) \wedge \neg AbRoot(p) \wedge (E_{Par_p} \neq B)$
 $FCorrection(p)$   $\equiv (S_p \neq idle) \wedge (E_p = F) \wedge \neg AbRoot(p) \wedge (E_{Par_p} = C)$
 $CCorrection(p)$   $\equiv (S_p \neq idle) \wedge (E_p = C) \wedge \neg AbRoot(p) \wedge (E_{Par_p} = F)$
 $CError(p)$        $\equiv FAbRoot(p) \vee BCorrection(p) \vee FCorrection(p) \vee CCorrection(p)$
Actions:
 $EB\text{-}action :: BError(p) \rightarrow E_p := B$;
 $EF\text{-}action :: FError(p) \rightarrow E_p := F$;
 $EC\text{-}action :: CError(p) \rightarrow E_p := C$; $S_p := idle$;
```

fair or unfair), the abnormal trees cannot prevent forever the progression of the visiting phase rooted at r. Then, the normal visiting phase terminates at r in a finite number of steps ($O(N^3)$). After this termination, the trace of the visiting phase are erased by the cleaning phase in $\Theta(N)$ steps. So, it is easy to see that, in the worst case, if the daemon tries to prevent r to initiate a new visiting phase, the system eventually reaches the normal initial configuration. In this configuration, r is the only enabled processor and *F-action* is the only enabled action at r. So, r executes *F-action* in the next step and we obtain a contradiction. Hence, from any configuration, the visiting phase starts at r after $O(N^3)$ steps.

Since r executes *F-action*, the visiting phase (rooted at r) sequentially progresses as deeply as possible in the network. When the visit cannot progress any more, the visit backtracks to the latest visited processor having some non-visited neighbors, if any. The visit terminates when it backtracks to r and r considers that its neighbourhood is completely visited. Obviously, to be DFS, the traversal performed by the visiting phase must not backtrack too earlier, i.e., the traversal must backtrack from p only when $\forall q \in Neig_p$, $q \in Tree(r)$. Now, this property is guaranteed by the confirmation phase. Indeed, since p hooks on to the normal tree ($Tree(r)$) by *F-action*, the confirmation phase insures p will executes $S_p = done$ (resp. *rdone* if $p = r$) only when $\forall q \in Neig_p$, $q \in Tree(r)$. Finally, we have already seen that the visiting phase rooted at r is executed in $O(N^3)$ steps.

Hence, by Definition 1, it is easy to see that Algorithm $\mathcal{DFS}$ is snap-stabilizing for Specification 1 under an unfair daemon (see [5] for a detailed proof).

Complexity Issues. From the previous explanations, we can deduce that the delay to start a DFS Wave is $O(N^3)$ steps and $O(N^2)$ rounds, respectively. Similarly, a complete DFS Wave is executed in $O(N^3)$ steps and $O(N^2)$ rounds, respectively. Consider now the space requirement. We do not make any bound on the value of the L variable but it is easy to see that Algorithm $\mathcal{DFS}$ remains valid if we bound the maximal value of L by N. So, by taking account of the other variables, we can deduce that Algorithm $\mathcal{DFS}$ is in $O(\Delta^2 \times N)$ states.

4 Conclusion

We proposed in [4] the first snap-stabilizing DFS wave protocol for arbitrary rooted networks assuming an unfair daemon. Until this paper, it was the only snap-stabilizing protocol solving this problem. Like in [4], the snap-stabilizing DFS wave protocol presented in this paper does not use any pre-constructed spanning tree and does not need to know the size of the network. Moreover, it is also proven assuming an unfair daemon. However, using this protocol, a complete DFS Wave is executed in $O(N^2)$ rounds and $O(N^3)$ steps while we obtain $O(N)$ rounds and $O(N^2)$ steps in [4] for the same task. But, our new solution brings some strong enhancements. In one hand, the new protocol works on a *semi-anonymous* network instead of a completely identified network. In the other hand, it requires $O(\Delta^2 \times N)$ states per processor instead of $O(N^N)$.

References

1. L Blin, A Cournier, and V Villain. An improved snap-stabilizing pif algorithm. In *DSN SSS'03 Workshop: Sixth Symposium on Self-Stabilizing Systems (SSS'03)*, pages 199–214. LNCS 2704, 2003.
2. A Bui, AK Datta, F Petit, and V Villain. State-optimal snap-stabilizing PIF in tree networks. In *Proceedings of the Fourth Workshop on Self-Stabilizing Systems*, pages 78–85, Austin, Texas, USA, June 1999. IEEE Computer Society Press.
3. A Cournier, AK Datta, F Petit, and V Villain. Enabling snap-stabilization. In *23th International Conference on Distributed Computing Systems (ICDCS 2003)*, pages 12–19, Providence, Rhode Island USA, May 19-22 2003. IEEE Computer Society Press.
4. A Cournier, S Devismes, F Petit, and V Villain. Snap-stabilizing depth-first search on arbitrary networks. In *OPODIS'04, International Conference On Principles Of Distributed Systems Proceedings*, pages 267–282. LNCS, 2004.
5. A Cournier, S Devismes, and V Villain. A snap-stabilizing dfs with a lower space requirement. Technical Report 2005-05, LaRIA, CNRS FRE 2733, 2004.
6. AK Datta, C Johnen, F Petit, and V Villain. Self-stabilizing depth-first token circulation in arbitrary rooted networks. *Distributed Computing*, 13(4):207–218, 2000.

7. EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.
8. S Dolev, A Israeli, and S Moran. Uniform dynamic self-stabilizing leader election. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):424–440, 1997.
9. ST Huang and NS Chen. Self-stabilizing depth-first token circulation on networks. *Distributed Computing*, 7:61–66, 1993.
10. C Johnen, C Alari, J Beauquier, and AK Datta. Self-stabilizing depth-first token passing on rooted networks. In *WDAG97 Distributed Algorithms 11th International Workshop Proceedings, Springer-Verlag LNCS:1320*, pages 260–274, Saarbrücken, Germany, September 24-26 1997. Springer-Verlag.
11. C Johnen and J Beauquier. Space-efficient distributed self-stabilizing depth-first token circulation. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 4.1–4.15, Las Vegas (UNLV), USA, May 28-29 1995. Chicago Journal of Theoretical Computer Science.
12. F Petit and V Villain. Time and space optimality of distributed depth-first token circulation algorithms. In *Proceedings of DIMACS Workshop on Distributed Data and Structures*, pages 91–106, Princeton, USA, May 10-11 1999. Carleton University Press.
13. G Tel. *Introduction to distributed algorithms.* Cambridge University Press, Cambridge, UK, Second edition 2001.