# Self-Stabilizing Depth-First Token Circulation In Arbitrary Rooted Networks *

Ajoy K. Datta,[1†] Colette Johnen,[2] Franck Petit,[3‡]
Vincent Villain[3‡]
[1] Department of Computer Science, University of Nevada, Las Vegas
[2] L.R.I./C.N.R.S., Université de Paris-Sud, France
[3] LaRIA, Université de Picardie Jules Verne, France

**Abstract:** We present a deterministic distributed depth-first token passing protocol on a rooted network. This protocol uses neither the processor identifiers nor the size of the network, but assumes the existence of a distinguished processor, called the root of the network. The protocol is self-stabilizing, meaning that starting from an arbitrary state (in response to an arbitrary perturbation modifying the memory state), it is guaranteed to reach a state with no more than one token in the network. Our protocol implements a strictly fair token circulation scheme. The proposed protocol has extremely small state requirement—only $3(\Delta + 1)$ states per processor, i.e., $O(log\Delta)$ bits per processor, where $\Delta$ is the degree of the network.

The protocol can be used to implement a strictly fair distributed mutual exclusion in any rooted network. This protocol can also be used to construct a DFS spanning tree.

**Keywords:** Distributed mutual exclusion, self-stabilization, spanning tree, token passing.

## 1 Introduction

Robustness is one of the most important requirements of modern distributed systems. Various types of faults are likely to occur at various parts of the system. These systems go through the transient states because they are exposed to constant change of their environment. The concept of self-stabilization [7] is the most general technique to design a system to tolerate arbitrary transient faults. A self-stabilizing system, regardless of the initial states of the processors and initial messages in the links, is guaranteed to converge to the intended behavior in finite time.

The depth-first token circulation problem is to implement a token circulating from one processor to the next in the depth-first order such that every processor gets the token at least once in every round (defined more formally later). In this paper, the token is initiated by the root of the network.

**Related Work.** Dijkstra introduced the property of self-stabilization in distributed systems by applying it to algorithms for mutual exclusion on a ring [7]. Several deterministic self-stabilizing token passing algorithms for different topologies have been proposed in the literature: [1, 4, 7, 11, 9] for a ring; [2, 10, 22] for a linear array of processors, and [16, 17, 21] for a tree network. Huang and Chen [13] presented a token circulation protocol for a connected network in non-deterministic

*A preliminary version of this work was presented at SIROCCO '98 [6].
†Supported in part by a sabbatical leave grant from University of Nevada, Las Vegas.
‡Supported in part by the Pole of Modelization of Picardie.

depth-first-search order, and Dolev, Israeli, and Moran [8] gave a mutual exclusion protocol on a tree network under the model whose actions only allow read/write atomicity.

One of the important performance issues of self-stabilizing algorithms is the memory requirement per processor. The memory requirement of a processor depends on the total number of states of the processor. The solution in [13] requires $O(n\Delta)$ states per processor, where $n$ is the number of processors. The algorithm in [8] constructs a spanning tree and implements a token circulation scheme on the constructed spanning tree. The spanning tree protocol uses a *distance* variable (which needs $n$ states) and the token circulation algorithm maintains a descendant pointer (which needs $\Delta$ states). So, the algorithm [8] requires at least $n\Delta$ states or $\log(n\Delta)$ bits.

A state-efficient token passing protocol on general network is presented in [15]. In this protocol, a processor $p_i$ needs to maintain $3(\Delta_i + 1)$ states ($\lceil \log(3(\Delta_i + 1)) \rceil$ bits), where $\Delta_i$ is the degree of $p_i$. Subsequently, this result was improved by Petit and Villain [18] to $2(\Delta_i + 1)$ states for a processor $p_i$. Both protocols do not use the *distance* variable. But, in these algorithms, a processor needs the knowledge of the state of the neighbors of its neighbors. Since the algorithms assume the atomic execution of the actions, this requirement makes the atomic step bigger—in one atomic step, a processor reads the state of its neighbors, the state of the neighbors of its neighbors, and finally changes its own state. This drawback has been removed in [14]. In this protocol, a processor only reads the state of its neighbors in an atomic step. Thus, this algorithm has a smaller atomicity than that in [15, 18]. The state requirement of this protocol is $12(\Delta_i + 1)$ states for a processor $p_i$. Petit and Villain [20] and [19] adapted the result of [15] and [18], respectively, in the message passing model.

**Contributions.** In this paper, we present a self-stabilizing depth-first token circulation scheme on a general network with a distinguished root, called Algorithm $\mathcal{TC}$. Algorithm $\mathcal{TC}$ has all the desirable features of the algorithm in [14]. In addition, we reduced the state requirement for a processor $p$ to $3(\Delta_p + 1)$ states (only $2(\Delta_p + 1)$ states for the root). Also, our algorithm is simpler (less number of actions) than that in [14]. Algorithm $\mathcal{TC}$ implements a strictly fair circulation of token, i.e., while each processor $p$ is waiting for the token, every processor $q$ ($q \neq p$) gets the token at most $\Delta_q$ times. Hence, Algorithm $\mathcal{TC}$ can also be used to implement a strictly fair distributed mutual exclusion among the processors.

**Outline of the Paper.** We present a self-stabilizing depth-first token circulation scheme on a general network with a distinguished root. The token passing problem is formally defined in Section 2.2. The rest of the paper is organized as follows: In Section 2, we describe the distributed systems and the model in which our token circulation scheme is written, and give a formal statement of the token passing problem solved in this paper. In Section 3, we present the token passing protocol, and in the following section (Section 4), we give the proof of correctness of the protocol. The state complexity of the protocol is given in Section 5. Finally, we make concluding remarks in Section 6.

## 2 Preliminaries

In this section, we define the distributed systems and programs considered in this paper, and state what it means for a protocol to be self-stabilizing. We then present the statement of the token passing problem and its properties.

### 2.1 Self-Stabilizing System

**System.** A *distributed system* is an undirected connected graph, $S = (V, E)$, where $V$ is a set of processors ($|V| = n$) and $E$ is the set of bidirectional communication link. We consider networks which are *asynchronous* and *rooted*, i.e., all processors, except the root are *anonymous*. We denote the root processor by $r$. The numbers, $1..n$, are used to identify the processors to present our ideas here, but no processor, except the root (identified by $r$), has any identity. A communication link $(p, q)$ exists iff $p$ and $q$ are neighbors. Every processor $p$ (including $r$) 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 order $\succ_p$. The number of neighbors of $p$, $|N_p|$, is called the *degree* of $p$ and is denoted by $\Delta_p$. We assume that $N_p$ is maintained by an underlying protocol.

**Programs.** Each processor executes the same program except the root $r$. The program consists of a set of *shared variables* (henceforth referred to as variables) and a finite set of actions. A processor can only write to its own variables and can only read its own variables and variables owned by the neighboring processors. So, the variables of $p$ can be accessed by $p$ and its neighbors.

Each action is uniquely identified by a label and is of the following form:

$$< label >:: < guard > \longrightarrow < statement >$$

The guard of an action in the program of $p$ is a boolean expression involving the variables of $p$ and its neighbors. The statement of an action of $p$ updates zero 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: the evaluation of a guard and the execution of the corresponding statement of an action, if executed, are done in one atomic step. The atomic execution of an action of $p$ is called a *step* of $p$.

The *state* of a processor is defined by the values of its variables. The *state* of a system is a product of the states of all processors ($\in V$). In the sequel, we refer to the state of a processor and system as a (*local*) *state* and *configuration*, respectively. Let a distributed protocol $\mathcal{P}$ be a collection of binary transition relations denoted by $\mapsto$, on $\mathcal{C}$, the set of all possible configurations of the system. A *computation* of a protocol $\mathcal{P}$ is a *maximal* sequence of configurations $e = (\gamma_0, \gamma_1, ..., \gamma_i, \gamma_{i+1}, ...)$, such that for $i \geq 0, \gamma_i \mapsto \gamma_{i+1}$ (a single *computation step*) if $\gamma_{i+1}$ exists, or $\gamma_i$ is a terminal configuration. *Maximality* means that the sequence is either infinite, or it is finite and no action of $\mathcal{P}$ is enabled in the final configuration. All computations considered in this paper are assumed to be maximal. During a computation step, one or more processors execute a step and a processor may take at most one step. This execution model is known as the *distributed daemon* [3]. We use the notation $Enable\,(A, p, \gamma)$ to indicate that the guard of the action $A$ is true at processor $p$ in the configuration $\gamma$. A processor $p$ is said to be *enabled* at $\gamma$ ($\gamma \in \mathcal{C}$) if there exists an action $A$ such that $Enable\,(A, p, \gamma)$. We assume a *weakly fair* daemon, meaning that if a processor $p$ is continuously *enabled*, then $p$ will be eventually chosen by the daemon to execute an action.

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 configuration $\beta$ is *reachable* from $\alpha$, denoted as $\alpha \leadsto \beta$, if there exists a computation $e = (\gamma_0, \gamma_1, \ldots, \gamma_i, \gamma_{i+1}, \ldots) \in \mathcal{E}_\alpha(\alpha = \gamma_0)$ such that $\beta = \gamma_i (i \geq 0)$. We will also use the notation $\alpha \leadsto_e \beta$ to indicate that $\beta$ is *reachable* from $\alpha$ during the particular computation $e$.

**Predicates.** 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 $\mathcal{X}$. A predicate is non-empty if there exists at least one element that satisfies the predicate. We define a special predicate $\mathsf{true}$ as follows: *for any* $x \in \mathcal{X}$, $x \vdash \mathsf{true}$.

**Self-Stabilization.** We use the following term, *attractor* in the definition of self-stabilization.

**Definition 2.1 (Attractor)** *Let $X$ and $Y$ be two predicates of a protocol $\mathcal{P}$ defined on $\mathcal{C}$ of system $\mathcal{S}$. $Y$ is an attractor for $X$ if and only if the following condition is true:*
$\forall \alpha \vdash X : \ \forall e \in \mathcal{E}_\alpha : \ e = (\gamma_0, \gamma_1, ...) :: \ \exists i \geq 0, \forall j \geq i, \gamma_j \vdash Y$. *We denote this relation as* $X \triangleright Y$.

**Definition 2.2 (Self-stabilization)** *The protocol $\mathcal{P}$ is self-stabilizing for the specification $\mathcal{SP}_\mathcal{P}$ on $\mathcal{E}$ if and only if there exists a predicate $\mathcal{L}_\mathcal{P}$ (called the legitimacy predicate) defined on $\mathcal{C}$ such that the following conditions hold:*

1. $\forall \alpha \vdash \mathcal{L}_\mathcal{P} : \ \forall e \in \mathcal{E}_\alpha :: \ e \vdash \mathcal{SP}_\mathcal{P}$ *(correctness).*

2. $\mathsf{true} \triangleright \mathcal{L}_\mathcal{P}$ *(closure and convergence).*

### 2.2 Specification of the Depth-First Token Passing Protocol

**Definition 2.3 (Token Circulation Round)** *We define a computation in the protocol $\mathcal{TC}$ starting from a configuration $\delta$ as a token circulation round (denoted as cround) if the following conditions are true:*

[S] *Exactly one processor holds a token in any configuration.*

[L1] *$r$ holds a token in $\delta$ and eventually, passes it to a processor.*

[L2] *When a processor p receives a token, p sends the token to a processor following the depth-first search order.*

Condition [S] defines the *safety* property, whereas Conditions [L1] and [L2] define the *liveness* property.

We are now ready to define the specification, $\mathcal{SP}_\mathcal{TC}$ of the protocol $\mathcal{TC}$. We consider a computation $e$ of $\mathcal{TC}$ to satisfy $\mathcal{SP}_\mathcal{TC}$ iff e is an infinite repetition of token circulation rounds.

## 3 Depth-First Token Passing Algorithm

In this section, we propose the self-stabilizing depth-first token circulation algorithm. We first present the data structure used by the processors. Then we present the formal algorithm. Next, we define some terms to be used later in the paper. We then explain the process of token circulation, followed by the method of error correction. In particular, we do not use the distance variable used in [13] to destroy the cycles. We use a method similar to the one introduced in [15] to remove the cycles in the network.

### 3.1 Data Structures and Algorithm $\mathcal{TC}$

To distinguish each token round, each processor $p$ uses a variable $C_p$, called the *round color*, which contains a value $\in \{0,1\}$ when the system is stabilized. A third color $E$, called the *Error* color, is used by processors, except the root, during the stabilization. In our token circulation scheme, we do not maintain any permanent descendant. After the token is passed from a processor $p$ to another processor $q$, we consider $p$ as the *parent* and $q$ the "current descendant" (or "current child") of $p$. But, when the token comes back from $q$ to $p$, we do not maintain the descendant relationship between $p$ and $q$. For the sake of simplicity, we will refer to the "current descendant" as simply the "descendant". The descendant relationship is indicated by the variable $D_p$ ($D_p \in N_p \cup \{\perp\}$). Each processor $p$ chooses its descendant using the local link ordering $\succ_p$ (see Section 2.1).

The self-stabilizing depth-first token circulation algorithm (Algorithm $\mathcal{TC}$) is shown in Algorithm 3.1 for the root $r$ and in Algorithm 3.2 for the other processors. To make the exposition clear, we present it in four groups: the *variables*, *actions*, *predicates*, and *macros*. The macros are not variables and are dynamically evaluated. $Par_p$ denotes the set of parents of $p$, i.e., $Par_p = \{q \in N_p \mid D_q = p\}$. $UV_p$ is the set of neighbors not visited by the token. $Search_p$ chooses the next neighbor from $UV_p$. In the following, $\sharp Par_p$ denotes the current number of parents of $p$. If $\sharp Par_p = 1$, then the only parent of $p$ is denoted as $MyP_p$ ($MyP_p$ is not used when $\sharp Par_p \neq 1$). The predicates are used to describe the guards of the actions in Algorithm $\mathcal{TC}$. In both Algorithms 3.1 and 3.2, Actions $TC1$ and $TC2$ implement the *token circulation*, i.e., the correct behavior of the system. The token circulates in the network according to Definition 2.3. Actions $EC1$, $EC2$, and $EC3$ of Algorithm 3.2 implement the *error correction* of the system; they are used to bring the system from an illegitimate configuration to a legitimate one. All these predicates will be explained in detail in Section 3.2.

---

**Algorithm 3.1** ($\mathcal{TC}$) For the root ($p = r$).

**Variables**

$C_p \;:\; C_p \in \{0,1\}$
$D_p \;:\; D_p \in N_p \cup \{\perp\}$

**Actions**

$TC1 \;::\; Forward(p) \longrightarrow C_p := (C_p + 1)\ mod\ 2;\ D_p := Search_p;$
$TC2 \;::\; Backtrack(p) \longrightarrow D_p := Search_p;$

**Predicates**

$Forward(p) \equiv (D_p = \perp)$
$Backtrack(p) \equiv (D_p \neq \perp) \wedge (D_{D_p} = \perp) \wedge (C_{D_p} = C_p)$

**Macro**

$UV_p = \{q \in N_p : (q \succ_p D_p) \wedge (C_q \neq C_p) \wedge ((C_q \neq E) \vee (D_q \neq \perp))\}$
$Search_p = \min_{\succ_p}(UV_p)$ **if** $(UV_p \neq \emptyset)$, $\perp$ **otherwise**

---

When the system stabilizes, the system must contain only one token which circulates in the DFS order. In such a configuration, a processor can make a move only if it holds the token. Holding the token means either $Forward(p)$ or $Backtrack(p)$ is true. Formally:

$$Token(p) \equiv Forward(p) \vee Backtrack(p)$$

**Algorithm 3.2** ($\mathcal{TC}$) For the other processors ($p \neq r$).

**Variables**

$C_p$ : $C_p \in \{0, 1, E\}$
$D_p$ : $D_p \in N_p \cup \{\bot\}$

**Actions**

$TC1$ :: $Forward(p) \longrightarrow C_p := (C_p + 1)\ mod\ 2;\ D_p := Search_p;$
$TC2$ :: $Backtrack(p) \longrightarrow D_p := Search_p;$
$EC1$ :: $Break(p) \longrightarrow D_p := \bot;$
$EC2$ :: $EDetect(p) \longrightarrow C_p := E;$
$EC3$ :: $EEnd(p) \longrightarrow$ **if** $(\sharp Par_p = 0)$ **then** $C_p := 0;$ **else** $C_p := C_r;$

**Predicates**

$$Forward(p) \equiv (D_p = \bot) \wedge (\sharp Par_p = 1) \wedge (C_p \neq E) \wedge (C_{MyP_p} = (C_p + 1)\ mod\ 2)$$
$$Backtrack(p) \equiv (D_p \neq \bot) \wedge (D_p \neq r) \wedge (D_{D_p} = \bot) \wedge (C_{D_p} = C_p) \wedge (C_p \neq E) \wedge (\sharp Par_p = 1)$$
$$BrkA(p) \equiv (D_p = r)$$
$$BrkB(p) \equiv (C_p = E) \wedge (\sharp Par_p > 1) \wedge (C_{D_p} = E)$$
$$BrkC(p) \equiv (C_p = E) \wedge (D_{D_p} = \bot)$$
$$BrkD(p) \equiv (C_p \neq E) \wedge (D_{D_p} = \bot) \wedge (\sharp Par_p = 0) \wedge ((C_{D_p} = E) \vee (C_{D_p} \neq (C_p + 1)\ mod\ 2))$$
$$Break(p) \equiv (D_p \neq \bot) \wedge (BrkA(p) \vee BrkB(p) \vee BrkC(p) \vee BrkD(p))$$
$$EDctA(p) \equiv (D_p \neq \bot) \wedge (C_{D_p} = E) \wedge (\sharp Par_p = 1)$$
$$EDctB(p) \equiv (D_p \neq r) \wedge (\sharp Par_p > 1)$$
$$EDetect(p) \equiv (C_p \neq E) \wedge (EDctA(p) \vee EDctB(p))$$
$$EEnd(p) \equiv (C_p = E) \wedge (D_p = \bot) \wedge ((\sharp Par_p = 0) \vee (Par_p = \{r\}))$$

**Macro**

$$Par_p = \{q \in N_p : D_q = p\}$$
$MyP_p = q \in Par_p$ **if** $Par_p = \{q\}$, $\bot$ **otherwise**
$$UV_p = \left\{ q \in N_p : \left( \begin{array}{l} (q \succ_p D_p) \wedge (C_q \neq C_p) \wedge (D_q \neq p) \wedge (q \neq r) \\ \wedge ((C_q \neq E) \vee (D_q \neq \bot)) \end{array} \right) \right\}$$
$Search_p = \min_{\succ_p}(UV_p)$ **if** $(UV_p \neq \emptyset)$, $\bot$ **otherwise**

## 3.2 Informal Explanation of Algorithm $\mathcal{TC}$

The proposed algorithm has two major tasks: (i) to circulate the token in the network in a deterministic depth-first order and (ii) to handle the abnormal situations (illegal configurations) due to the unpredictable initial configurations and transient errors. The tasks (i) and (ii) are explained with examples in Paragraphs **Token Circulation** and **Error Correction**, respectively.

### 3.2.1 Token Circulation.

The root $r$ initiates the token circulation round. The token then traverses all processors during a token circulation round (Definition 2.3).

We use $\delta_0 \in \mathcal{C}$ to denote a configuration where every processor in the system is path-free and has the color 0. Similarly, $\delta_1$ denotes the configuration in which every processor is path-free and has the color 1. That is,

$\delta_0 \equiv \forall p \in V :: Par_p = \emptyset \wedge D_p = \bot \wedge C_p = 0$
$\delta_1 \equiv \forall p \in V :: Par_p = \emptyset \wedge D_p = \bot \wedge C_p = 1.$

Both $\delta_0$ and $\delta_1$ are among the possible configurations from where the algorithm behaves correctly, i.e., starting from $\delta_0$ (respectively, from $\delta_1$), Algorithm $\mathcal{TC}$ circulates the token (represented by the

predicate $Token()$) in the depth-first search order to reach $\delta_1$ (respectively, $\delta_0$). This is called one cround (see Definition 2.3). The *cround* leading the system from $\delta_1$ to $\delta_0$ (respectively, from $\delta_0$ to $\delta_1$ is said to be 0-colored (respectively, 1-colored), i.e., starting from a configuration where the color of every processor is 1 (respectively, 0), the *cround* eventually colors every processor in 0 (respectively, 1). The token circulation consists of successive *crounds*, alternately colored with 0 and 1. After stabilization, the system repeats the *crounds* forever. The *cround* is implemented by Actions $TC1$ and $TC2$. Every suffix of the computation starting from $\delta_0$ or $\delta_1$ satisfies the specification $\mathcal{SP}_{\mathcal{TC}}$.

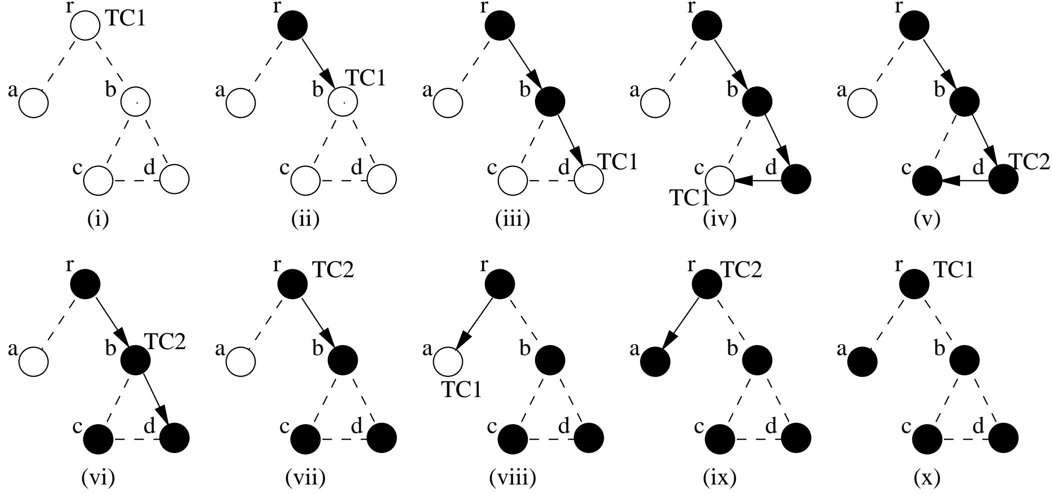


Figure 3.1: Depth-First Search Token Circulation.

Consider the example in Figure 3.1. Configuration (i) corresponds to the configuration $\delta_0$. In this configuration, $Forward(r)$ is true and the only process *enabled* is $r$ and the only action enabled at $r$ is $TC1$. The root changes its color ($C_r := (C_r + 1)\ mod\ 2$) and chooses the processor $b$ as the descendant (*Search* predicate). This is shown in Configuration (ii). Similarly, $b$ changes its color and chooses a descendant (Configuration (iii)). This process of extending the path continues until $c$ executes Action $TC1$. $c$ does not have any neighbor to choose from. So, $c$ executes $D_c := \perp$ (*Search*). This indicates to its parent $d$ that the token has traversed all processors reachable from $c$ in the DFS tree (Configuration (v)). Now, $Backtrack(d)$ becomes true and $d$ can execute $TC2$. Since $d$ has no more unvisited neighbors, $D_d$ becomes equal to $\perp$ (Configuration (vi)). Actions $TC1$ and $TC2$ are repeated until all processors are visited by the token (Configurations (vii) to (x)). Configuration (x) corresponds to $\delta_1$. Now, $r$ changes its color to 0 and starts a new round with this color.

### 3.2.2 Error Correction.

We now consider the transient failures. An illegitimate configuration is shown in Figure 3.2.

**Some Definitions.** A path $\mu_p$ is a sequence $(p_1, p_2, \ldots, p_l)$ such that $(i)$ $p = p_1$, $(ii)$ $l \geq 2$, $(iii)$ $\forall i \in [1, l-1]$, $D_{p_i} = p_{i+1}$, and $(iv)$ $D_{p_l} = \perp$ or $D_{p_l} \in \{p_1, p_2, \ldots, p_{l-1}\}$. $\forall i \in [1, l], p_i$ is said to *belong to* the path $\mu_p$ and is denoted as $p_i \in \mu_p$.

If $Par_p = \emptyset$, then $\mu_p$ is called a *rooted path* (the path is rooted at $p$). A path $\mu_p$ rooted at $p \neq r$ is called an *illegal rooted path* and $p$ is called an *illegal root.* A path $\mu_p$ rooted at $r$ is called a *legal (rooted) path.*

The processor $p_l \in \mu_p$ is called a *leaf* if $D_{p_l} = \bot$. The leaf of a legal (respectively, illegal) rooted path is called *legal* (respectively, *illegal*) *leaf*. A leaf $p'$ is termed as a *live* (respectively, *dead*) *leaf*, if $C_{p'} \neq E$ (respectively, $C_{p'} = E$). A rooted path with a live leaf is termed as a *live rooted path*. All other rooted paths are called *dead rooted paths*.

The path $\mu_p$ is called a *cycle* if $D_{p_l} \in \{p_1, p_2, \ldots, p_{l-1}\}$. A cycle $\mu_p$ is called a *strict cycle* if $\forall i \in [2, l]$, $p_{i-1}$ is the only parent of $p_i$ and $p_l$ is the only parent of $p_1$. If there exists at least one rooted path $\mu_q$ such that all processors in a cycle $\mu_p$ belong to $\mu_q$, i.e., $\exists p_i \in \mu_p$ such that $i \in [2, l]$ and $\sharp Par_{p_i} > 1$, then the rooted path $\mu_p$ is called a *rooted cycle*.

Every processor $p$ such that $Par_p = \emptyset$ and $D_p = \bot$ is called *path-free*, meaning $p$ does not belong to any path.

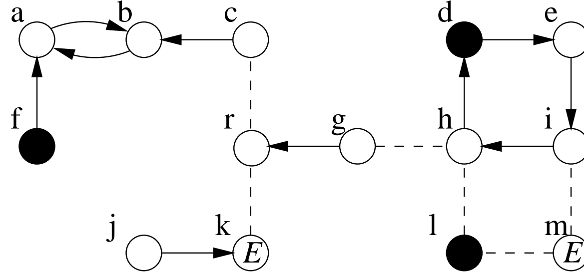


Figure 3.2: A Possible Configuration.

These definitions are illustrated in Figure 3.2. Processors $c, f, g$, and $j$ are illegal roots. $d, e, i, h$ form a strict cycle. $f$ and $c$ are roots of a rooted cycle. $k$ is a dead leaf. $l$ and $m$ are path-free.

**Root Without Parent.** Consider the illegal configuration in which $r$ has parents. For every parent $p$ of $r$, $p$ satisfies $BrkA$ ($D_p = r$) and hence, $Break(p)$. Upon executing Action $EC1$, $p$ eventually destroys the descendant pointer to $r$ and since, $r$ cannot be chosen as a descendant in the algorithm (see macro $UV_p$), $r$ eventually will not be on any illegal paths.

**Cycle Destruction.** We first explain how the strict cycles are destroyed by the token circulation mechanism. Our strategy is based on the following three ideas.

The first idea, introduced in [15], is based on the following observation: If a *cround* starts with a round color $c$ ($\in \{0, 1\}$) in a correct initial configuration (i.e., in $\delta_0$ or $\delta_1$), every neighbor of the legal leaf is either

($i$) path-free and in color $(c + 1)$ *mod* 2, or

($ii$) is in the legal path, i.e., has a descendant and is $c$-colored.

Assume that the system is in an abnormal configuration. In such a configuration, there may exist a neighbor $p$ of the legal leaf (say, $q$) such that $p$ has a descendant ($D_p \neq \bot$), $p$ is not $c$-colored (i.e., $p$ is in either $(c + 1)$ *mod* 2 or *Error*), and $\mu_p$ forms a strict cycle. Since $p$ does not have the same color as $q$, following the "normal" behavior of the algorithm (i.e., $q$ chooses neighbors not having the same color as its own), $q$ eventually chooses $p$ as the descendant. Then, the path $\mu_p$ becomes the end of the legal path. As explained in details below, this allows to break cycles since $\mu_p$ could be a strict cycle.

The second idea is introduced in [18]. In [13] and [15], the color *Error* is used to destroy the illegal rooted paths by the illegal root. But, in [18], *Error* color is utilized to avoid the progress

of illegal leaves created by the removal of cycles. Illegal rooted path removal is explained in more details later.

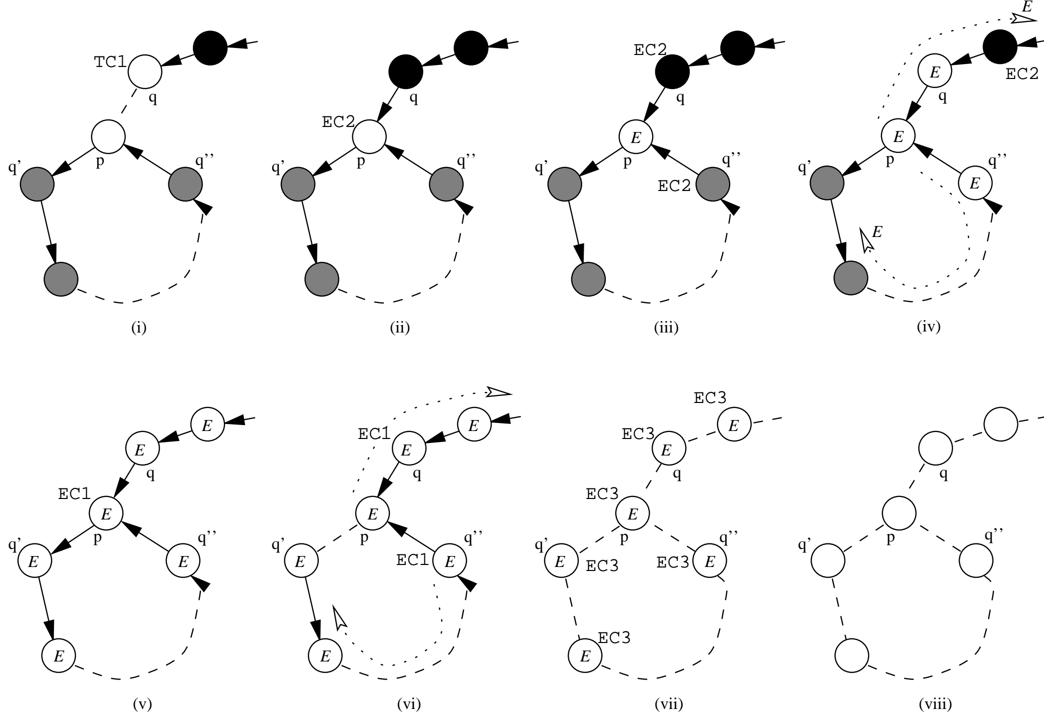


Figure 3.3: Cycles Destruction.

We now explain the third idea (introduced in this paper) using the example in Figure 3.3. Without loss of generality, assume that in Figure 3.3, black processors (respectively, white processors) are 0-colored (respectively, 1-colored) and grey processors can have any color (in $\{0, 1, E\}$). Note that the case where the color of $p$ is $E$, is similar. In that case, the color of $q$ can be either 0 or 1.

Let us now consider strict cycles using Figure 3.3. Our algorithm ensures that a neighbor ($q$ in figure) of a processor ($p$ in figure) involved in a strict cycle (the processors $p$, $q'$, ..., $q''$ form the cycle) eventually becomes the leaf of a rooted path. Since the token circulation consists of successive *crounds*, alternately colored with 0 and 1, $q$ has to eventually choose $p$ as the descendant. In Configuration (i), $p$ is the first descendant chosen by $q$ executing Action $TC1$ (i.e., $p = \min_{\succ_q}(UV_q)$ in Macro $Search_q$). Note that the case where $p$ is not the first descendant of $q$ is similar. In that case, our algorithm ensures that $q$ eventually chooses $p$ as the descendant after at most $\Delta_q - 2$ times execution of $TC2$. Furthermore, it is shown in Section 4.5 that if the situation described above does not involve an illegal leaf, then at least the legal leaf eventually reaches a processor involved in a strict cycles which is not yet destroyed.

In Configuration (i), $q$ chooses $p$ as the descendant (Configuration (ii)). After being chosen as a child, $p$ detects that it has more than one parent ($EDctB(p)$ is true) and executes $EC2$ to become a $E$-colored processor (Configuration (iii)).

The key point of our strategy is that the color $E$ is propagated from a descendant to its parents. In our example, for each parent $p'$ of $p$ (i.e., $q$ and $q''$), $EDctA(p')$ is true. The parents of the $E$ colored processors execute $EC2$ to propagate the color $E$ along all the paths $p'$ belongs to (Configurations

(iii) and (iv)).

Since $p$ belongs to a cycle, its descendant ($q'$ in the figure) is eventually $E$-colored (Configuration (v)). $p$ then satisfies $BrkB$, executes $EC1$, and detaches $q'$ to break the cycle (Configuration (vi)). Now, the cycle is replaced by two rooted paths that end at a dead leaf. Next, for every parent $p' \neq r$ of $p$, either $BrkD(p')$ or $BrkC(p')$ becomes true depending on if $p'$ is an illegal root or not a root, respectively. The parent of the dead leaf satisfies either $BrkD$ or $BrkC$, and eventually executes Action $EC1$ (Configuration (vii)). Finally, the $E$-colored, path-free processors are made 0-colored by Action $EC3$ (Configuration (viii)).

It is easy to observe that the rooted cycles can be destroyed using the same mechanism as above (Configurations (ii)-((viii)).

**Illegal Rooted Path Removal.** The protocol must also destroy all illegal rooted paths. If a rooted path $\mu_p$ is a rooted cycle, it is destroyed using the cycle destruction mechanism described above. Otherwise, it has a dead ($E$-colored) or a live (not $E$-colored) leaf. In the first case, $\mu_p$ is self-destroyed as above (Configurations (vi) -(viii)). In the second case, $\mu_p$ is self-destroyed by just allowing the token to circulate. The macro $Search_p$ uses the local ordering among the neighbors of a processor to avoid repeating a path during a token circulation round. Thus, a token circulation round cannot loop, and hence, eventually terminates. Once the current token circulation round completes, the processors which were in the illegal path become path-free. Moreover, the illegal root cannot initiate a new token circulation, i.e., cannot create a new path because only the root can initiate a token circulation round.

# 4 Correctness of the Token Passing Protocol $\mathcal{TC}$

We define the legitimacy predicate, $\mathcal{L}_{\mathcal{TC}}$ as follows: A configuration $\gamma$ satisfies $\mathcal{L}_{\mathcal{TC}}$ if it is reachable from $\delta_0$, i.e.,

$\mathcal{L}_{\mathcal{TC}} \equiv \delta_0 \rightsquigarrow \gamma$.

Note that we could also define $\mathcal{L}_{\mathcal{TC}}$ in terms of $\delta_1$ since $\delta_0 \rightsquigarrow \delta_1$ by Actions TC1 and TC2.

We apply the convergence stair method [12] to prove the closure and convergence of our protocol. We exhibit a finite sequence of state predicates $\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_m$, of Protocol $\mathcal{TC}$ such that the following conditions hold:

(i) $\mathcal{A}_0 \equiv true$ (meaning any arbitrary state)

(ii) $\forall j :\ 0 \leq j < m ::\ \mathcal{A}_j \triangleright \mathcal{A}_{j+1}$

(iii) $\mathcal{A}_m \Rightarrow \mathcal{L}_{TC}$

The proof outline is as follows:

In Section 4.1, we show that eventually no processor has the root as a descendant. Then, we prove that a locked processor (which never executes any action) cannot be the root, and either it does not have any descendant, or it belongs to a strict cycle (Section 4.2). This last result leads to the proof of liveness of the algorithm and is also used to prove that all illegal live rooted paths are eventually destroyed (Section 4.3). Once no illegal live rooted path exists, the system contains only one token. In Section 4.4, we show that since the root changes its color infinitely often, the legal path will be eventually colored with the color of the root. Then in Section 4.5, we prove that all cycles are eventually detected and destroyed. Finally, in Section 4.6, we prove that the system reaches a configuration which satisfies $\mathcal{L}_{\mathcal{TC}}$, and Protocol $\mathcal{TC}$ is self-stabilizing.

### 4.1 Root Without A Parent

In this section, we show that the system trivially reaches a configuration in which $r$ does not have any parent.

We define $\mathcal{A}_1 \equiv (\forall p \in V : D_p \neq r)$.

**Theorem 4.1** $\mathcal{A}_0 \triangleright \mathcal{A}_1$.

**Proof.** Let $Par_r$ be the set of processors $p$ such that $D_p = r$. Denote $\sharp Par_r$ the number of processors in $Par_p$.

$\mathcal{A}_1$ is closed: The root $r$ can not be chosen as a descendant by a process $p \neq r$ (see in macro $UV_p$). Hence, $\sharp Par_r$ cannot increase.

Every computation leads to $\mathcal{A}_1$: $(\forall p \in V, \forall \alpha \in \mathcal{C} :: D_p = r) \Rightarrow \mathit{Enable}\,(EC1, p, \alpha)$. $p$ executes $EC1$ in the configuration $\alpha$ or $\mathit{Enable}\,(EC1, p, \beta)$ where $\alpha \mapsto \beta$. By fairness, $\forall e \in \mathcal{E}_\alpha : \exists \beta : \alpha \rightsquigarrow_e \beta$ such that $p$ executes $EC1$ in $\beta$. Hence, $\sharp Par_r$ decreases. Since $\sharp Par_r$ cannot increase, $\forall e \in \mathcal{E}_\alpha$ : $\exists \beta : \alpha \rightsquigarrow_e \beta :: \beta \vdash (Par_r = \emptyset)$. □

### 4.2 Properties of Locked Processors

We need the following term throughout this section:

A processor $p$ is said to be *Locked* in a configuration $\alpha$, if in all configurations reachable from $\alpha$, $C_p$ and $D_p$ remain constant. Formally:

> $\mathit{Locked}(p, \alpha) \equiv (\forall \beta : \alpha \rightsquigarrow \beta :: (C_{p_\alpha} = C_{p_\beta}) \wedge (D_{p_\alpha} = D_{p_\beta})$, where $V_{p_\gamma}$ denotes the value of $V_p$ in the configuration $\gamma$)

Since the daemon is weakly fair, $\mathit{Locked}\,(p, \alpha)$ implies that $p$ is not continuously *enabled* in all configurations reachable from $\alpha$ and that $p$ never executes an action.

We first prove that if a processor $q$ is the descendant of a *Locked* processor $p$, then $q$ is eventually *Locked* (Lemma 4.2). Then we establish that if $p$ is not the root $r$, then $q$ has a descendant and will maintain the descendant forever (Lemma 4.3).

**Lemma 4.2** $\forall p, q \in V$, $\forall \alpha \vdash \mathcal{A}_1$ :
$(\mathit{Locked}(p, \alpha) \wedge (D_p = q)) \Rightarrow (\forall e \in \mathcal{E}_\alpha : \exists \beta : \alpha \rightsquigarrow_e \beta :: \mathit{Locked}(q, \beta))$.

**Proof.** We will prove this by contradiction. We assume the contrary, i.e.,
$\exists p, q \in V$, $\exists \alpha \vdash \mathcal{A}_1 : (\mathit{Locked}(p, \alpha) \wedge (D_p = q)) \wedge (\exists e \in \mathcal{E}_\alpha : \forall \beta : \alpha \rightsquigarrow_e \beta :: \neg \mathit{Locked}(q, \beta))$. Thus, $q$ executes an action infinity often in $e$. (Note that $\forall \alpha \in \mathcal{A}_1 : q \neq r$).

As $p$ is locked, $\forall \beta : \alpha \rightsquigarrow_e \beta :: p \in Par_q$ in $\beta$.

1. Assume that $\exists \beta : \alpha \rightsquigarrow_e \beta :: C_q = E$ in $\beta$. Since $C_q = E$, only the guards of Actions $EC1$ and $EC3$ can be true. Thus, $q$ can execute only $EC1$ or $EC3$ in $\beta$ or $\beta'$, where $\beta \rightsquigarrow_e \beta'$.

   a. Assume that $D_q = \perp$ in $\beta$. Then, $q$ can execute only $EC3$ in $\beta$. $Par_q$ does not increase while $q$ does not execute $EC3$ because no neighbor of $q$ can select $q$ (see macro $UV_p$). Since $q$ is not locked, $\exists \beta' : \beta \rightsquigarrow_e \beta' :: \mathit{Enable}(EC3, q, \beta')$. Since $p \in Par_q$, $\mathit{Enable}(EC3, q, \beta')$ implies $[p = r$ and $Par_q = \{r\}]$ in $\beta'$. Execution of $EC3$ makes $[C_p := C_r$ and $\mathit{Enable}(TC2, r, \beta')]$ ($\mathit{Backtrack}(r)$ is true). In all $\beta''$ such that $\beta' \rightsquigarrow_e \beta''$,

$Enable(TC2, p, \beta'')$ since $p$ does not execute an action. By fairness, $p$ eventually runs $TC2$ which contradicts the assumption, $Locked(p, \alpha)$. (This result is also valid (with respect to $TC2$) if $p = r$).

b. Assume that $D_q \neq \bot$ in $\beta$. Then, $q$ can execute $EC1$ only in $\beta$. Since $q$ is not locked, $\exists \beta' : \beta \rightsquigarrow_e \beta'$ such that $q$ eventually executes $EC1$ in $\beta'$. After the execution of $EC1$ by $q$, $D_q = \bot$. Thus, we arrive at the assumed state of Case 1a.

2. Assume that $\forall \beta : \ \alpha \rightsquigarrow_e \beta :: \ C_q \neq E$ in $\beta$. So, $\forall \beta : \ \alpha \rightsquigarrow_e \beta$, $Enable(EC3, q, \beta)$ is not satisfied. Similarly, $Enable(EC1, q, \beta)$ is not satisfied ($C_q \neq E$ and $\sharp Par_q > 0$ because $p \in Par_q$). Furthermore, $q$ cannot execute $EC2$ because otherwise, $\exists \beta : \ \alpha \rightsquigarrow_e \beta$ such that $C_q = E$, which contradicts the assumption. Therefore, $\forall \beta : \alpha \rightsquigarrow_e \beta$, $q$ cannot execute $EC1$, $EC2$, and $EC3$.

   a. Assume that $q$ executes $TC2$ infinitely often. Since $D_q$ strictly increases with respect to $\succ_q$ (see macro $Search_p$), $\exists \beta : \alpha \rightsquigarrow_e \beta :: \ D_q = \bot$ in $\beta$. In this case, $\forall \beta' : \beta \rightsquigarrow_e \beta'$, $Enable(TC2, p, \beta')$, $Enable(EC1, p, \beta')$, or $Enable(EC2, p, \beta')$ depending on $\sharp Par_p$ and $C_p$. By fairness, $p$ eventually executes $TC2$, $EC1$, or $EC2$, which contradicts the assumption, $Locked(p, \alpha)$.

   b. Assume that $q$ executes $TC2$ a finite number of times only. So, $\exists \beta : \alpha \rightsquigarrow_e \beta$ after which $q$ executes only $TC1$. But, after the execution of $TC1$, $C_q = C_p$. If $D_q = \bot$ in $\beta$ then, $\forall \beta' : \ \beta \rightsquigarrow_e \beta'$, $Enable(TC2, p, \beta')$, $Enable(EC1, p, \beta')$, or $Enable(EC2, p, \beta')$; by fairness, $p$ eventually executes an action which contradicts the assumption, $Locked(p, \alpha)$. If $D_q \neq \bot$ in $\beta$, then $q$ can only execute $TC2$, which contradicts the hypotheses.

$\Box$

**Lemma 4.3** $\forall p, q \in V$, $\forall \alpha \vdash \mathcal{A}_1$ :
$(Locked(p, \alpha) \wedge (D_p = q) \wedge (p \neq r)) \Rightarrow (\forall e \in \mathcal{E}_\alpha : \ \exists \beta : \ \alpha \rightsquigarrow_e \beta : \ \forall \beta' : \beta \rightsquigarrow_e \beta' :: \ D_q \neq \bot \ in \ \beta')$

**Proof.** By Lemma 4.2, $\forall e \in \mathcal{E}_\alpha : \ \exists \beta : \ \alpha \rightsquigarrow_e \beta :: \ Locked(q, \beta)$. So, $\forall \beta' : \ \beta \rightsquigarrow_e \beta'$, $D_q$ remains unchanged.

Assume $D_q = \bot$ in $\beta$. We will consider the following cases to arrive at the contradiction.

1. Assume that $C_p = E$ in $\beta$. Then, irrespective of the color of $q$, $\forall \beta' : \beta \rightsquigarrow_e \beta' :: \ Enable(EC1, p, \beta')$ ($C_p = E$ and $D_{D_p} = \bot$ in $\beta$). By fairness, $p$ eventually executes $EC1$. But, that is not possible since $p$ is locked.

2. Assume that $C_p \neq E$ in $\beta$.

   a. Assume that $C_q = E$ in $\beta$. Then $\forall \beta' : \beta \rightsquigarrow_e \beta'$, $Enable(EC1, p, \beta')$ or $Enable(EC2, p, \beta')$ (depending on the value of $\sharp Par_p$). By fairness, $p$ eventually executes either $EC1$ or $EC2$, which is not possible since $p$ is locked.

   b. Assume that $C_q = C_p$ in $\beta$. Then, $\forall \beta' : \ \beta \rightsquigarrow_e \beta'$, $Enable(TC2, p, \beta')$, $Enable(EC1, p, \beta')$, or $Enable(EC2, p, \beta')$ (if $\sharp Par_p = 1, 0$, or $> 1$, respectively). By fairness, $p$ eventually executes $TC2$, $EC1$, or $EC2$, which is not possible since $p$ is locked.

c. Assume that $(C_q \neq E) \wedge (C_q = (C_p + 1)\ mod\ 2)$ in $\beta$. If $\forall \beta' : \beta \rightsquigarrow_e \beta' ::\ \sharp Par_q = 1$ $(Par_q = \{p\})$ in $\beta'$, then $Enable(TC1, q, \beta')$. If $\exists \beta' : \beta \rightsquigarrow_e \beta' ::\ Par_q > 1$ $(Par_q \supseteq \{p\})$ in $\beta'$, then $\forall \beta'' : \beta' \rightsquigarrow_e \beta'' ::\ Enable(EC2, q, \beta'')$. By fairness, $q$ eventually executes either $TC1$ or $EC2$, both of which are not possible since $q$ is locked.

□

Now, we will prove the main results of this section. We first establish that a locked processor cannot be in a rooted cycle (Lemma 4.4). We prove this by contradiction. If $p$ belongs to a rooted cycle, then a processor $q$ in that cycle has at least two parents. The processor $q$ will eventually get the color $E$ and the color $E$ will be propagated along the cycle in the direction from the descendants towards the parents. So, $q$'s descendant will also be $E$-colored. Then, $q$ will break the cycle. By induction, every processor in the rooted cycle will eventually detach its descendant. Thus, $p$ is not *Locked*.

Then we show that a locked processor ($\neq r$) has no descendant or it belongs to a strict cycle (Theorem 4.5). Finally, we show that the root cannot be locked (Theorem 4.6), which implies the liveness of the algorithm.

**Lemma 4.4** $\forall p \in V,\ \forall \alpha \vdash \mathcal{A}_1 :\ Locked(p, \alpha) \Rightarrow$ *p does not belong to a rooted cycle.*

**Proof.** Let $\mu_q = (p_1, p_2, ..., p_{l-1}, p_l)$ be a rooted cycle where $p \in \mu_q$ and $D_{p_l} = p_i,\ i \in [2, l-1]$ $(\sharp Par_{p_i} > 1)$. We will prove this lemma by contradiction by assuming the contrary, i.e., there is a processor p such that $Locked(p, \alpha)$ is true and $p$ belongs to the rooted cycle $\mu_q$.

1. Assume that $p = p_i$ and $C_{p_i} \neq E$. Then $EDetect(p_i)$ is true. Thus, $Enable(EC2, p_i, \alpha)$ and $\forall \beta : \alpha \rightsquigarrow \beta$, $Enable(EC2, p_i, \beta)$ remains true because no action allows the parent of $p_i$ to detach it because $D_p \neq \perp$. By fairness, $p_i$ eventually executes $EC2$ which contradicts the assumption ($p$ is *Locked*).

2. Assume that $p = p_i$ and $C_p = E$. Since $p$ is *Locked* (according to our assumption), by Lemma 4.2, all processors $p_j, j \in [i, l]$, also are *Locked*.

   a. Assume that $\exists j \in [i+1, l] :: C_{p_j} \neq E$, $C_{p_{j+1}} = E$. Then $EDetect(p_j)$ is true. Thus, $Enable(EC2, p_j, \alpha)$, and $\forall \beta : \alpha \rightsquigarrow \beta$, $Enable(EC2, p_j, \beta)$ remains true while $p_j$ does not execute $EC2$, which contradicts the assumption, $p_j$ is *Locked*.

   b. Assume that $\forall j \in [i, l] ::\ C_{p_j} = E$. Then $Break(p_i)$ is true. Thus, $Enable(EC1,\ p_i, \alpha)$ is true and remains true while $p_i$ does not execute $EC1$, which contradicts our assumption, $p_i$ is *Locked*.

3. Assume that $p \neq p_i$. From Cases 1, 2a, and 2b, $p_i$ cannot be *Locked*.

   a. Assume that $p = p_j, j \in [i+1, l]$. Then by Lemma 4.2, all processors $p_k, k \in [i, l]$, are also *Locked*, which contradicts the fact that $p_i$ is not *Locked*.

   b. Assume that $p = p_j, j \in [1, i-1]$. Then by Lemma 4.2, all processors $p_k, k \in [j+1, l]$, are also *Locked*, which contradicts the fact that $p_i$ is not *Locked* $(i \in [j+1, l])$.

□

**Theorem 4.5** $\forall p \in V,\ \forall \alpha \vdash \mathcal{A}_1$ :
$(Locked(p,\alpha) \wedge (p \neq r)) \Rightarrow ((D_p = \perp \textit{ in } \alpha) \vee (\forall e \in \mathcal{E}_\alpha :\ \exists \beta :\ \alpha \rightsquigarrow_e \beta :: \textit{ p belongs to a strict cycle}))$.

**Proof.** Assume that $D_p = q$ $(q \neq r)$. By Lemmas 4.2 and 4.3, $\forall e \in \mathcal{E}_\alpha :\ \exists \beta : \alpha \rightsquigarrow_e \beta ::$ $Locked(q,\beta)$ and $D_q \neq \perp$ in $\beta$. By induction, the descendant of $q$ will also be eventually locked, and so on. Since the graph $S$ is finite, $p$ belongs to a cycle. By Lemma 4.4, $p$ cannot belong to a rooted cycle. Thus, $p$ belongs to a strict cycle. □

**Theorem 4.6** $\forall p \in V,\ \forall \alpha \vdash \mathcal{A}_1 :\ Locked(p,\alpha) \Rightarrow (p \neq r)$.

**Proof.** We will prove this theorem by contradiction. Assume that $\exists \alpha \vdash \mathcal{A}_1 ::\ Locked(p,\alpha) \wedge p = r$.

1. Assume that $r$ has no descendant. Then, $\forall \beta : \alpha \rightsquigarrow \beta ::\ Enable(TC1, r, \beta)$. Thus, by fairness, $r$ is not *Locked*.

2. Assume that $r$ has a descendant, $q$. Then, by Lemma 4.2, $\exists \beta : \alpha \rightsquigarrow \beta ::\ q$ is also *Locked*. So, by Theorem 4.5, either $q$ exists in a strict cycle or $D_q = \perp$ in $\beta$.

   a. $D_q = \perp$ in $\beta$.
      (i) Assume that $\sharp Par_q = 1$ $(Par_q = \{r\})$ in $\beta$.
         (1) $C_q = E$ in $\beta$. Then, $\forall \beta' : \beta \rightsquigarrow \beta' ::\ Enable(EC3,\ q, \beta')$. So, by fairness, $q$ is not *Locked*.
         (2) $C_q \neq E$ in $\beta$.
         If $\exists \beta' : \beta \rightsquigarrow \beta' ::\ Par_q \supsetneq \{r\}$ (i.e., $q$ has been chosen as the descendant by a neighbor $\neq r$), then $\forall \beta'' : \beta' \rightsquigarrow \beta'' ::\ Enable(EC2, q, \beta'')$ and $\forall q' \in Par_q$, $q'$ cannot change $D_{q'}$. So, by fairness, $q$ is not *Locked* ($q$ eventually executes $EC2$).
         If $\forall \beta' : \beta \rightsquigarrow \beta' ::\ Par_q = \{r\}$, then $\forall \beta' : \beta \rightsquigarrow \beta' ::\ Enable(TC1, q, \beta') \vee Enable(TC2, r, \beta')$. Thus, by fairness, either $q$ or $r$ is not *Locked*.
      (ii) Assume that $\sharp Par_q > 1$ $(Par_q \supsetneq \{r\})$ in $\beta$.
         (1) $\exists \beta' : \beta \rightsquigarrow \beta' ::\ \sharp Par_q = 1$ in $\beta'$. Thus, $Par_q = \{r\}$ in $\beta'$ and by Case 2a(i), this is not possible.
         (2) $\forall \beta' : \beta \rightsquigarrow \beta' ::\ \sharp Par_q > 1$ in $\beta'$.
         If $C_q \neq E$ in $\beta$, then $\forall \beta' : \beta \rightsquigarrow \beta' :: Enable(EC2, q, \beta')$. So, by fairness, $q$ is not *Locked*.
         If $C_q = E$ in $\beta$, then the parents of $q$ $(\neq r)$ are continuously enabled to execute Actions $EC1$ or $EC2$ until $q$ remains their descendant. But, these parents of $q$ can execute $EC2$ at most once (to get the color $E$). After the execution of $EC2$, the parents of $q$ are continuously enabled to execute $EC1$. Since $C_q = E$ and $D_q = \perp$, $q$ cannot get a new parent. Thus, after repeated execution of $EC1$, eventually, $q$ will have no parents except $r$. This contradicts the assumption, $\forall \beta' : \beta \rightsquigarrow \beta' ::\ \sharp Par_q > 1$ in $\beta'$.

   b. $q$ has a descendant and is inside a strict cycle in $\beta$. Since $\forall \alpha \vdash \mathcal{A}_1$, $r$ has no parent, $q$ must belong to a rooted cycle (Theorem 4.1), which contradicts the assumption.

□

**Corollary 4.7** *In any configuration* $\vdash \mathcal{A}_1$*, at least one processor is enabled.*

### 4.3 Destruction of Live Illegal Rooted Paths

In this section, we show that all live illegal rooted paths are eventually destroyed.

The next result follows from Theorem 4.5.

**Lemma 4.8** *No processor can stay in a live illegal rooted path forever.*

**Corollary 4.9** *Every illegal root eventually executes Action EC1.*

Let us denote the number of live illegal leaves by $LIL$. In the next Lemma (Lemma 4.10), we prove that Algorithm $\mathcal{TC}$ cannot create a new live illegal leaf.

**Lemma 4.10** *$\forall \alpha \vdash \mathcal{A}_1$, $\forall \beta$ such that $\alpha \mapsto \beta$, the value of $LIL$ in $\beta$ is less than or equal to that of $LIL$ in $\alpha$.*

**Proof.** Assume the contrary, i.e., $LIL$ in $\beta$ is greater than $LIL$ in $\alpha$. Then one of the following is true: (1) A dead illegal leaf becomes a live illegal leaf, (2) A path is broken creating a live illegal leaf, and (3) A processor, other than the root, becomes the root of an illegal rooted path.

1. For any $p$ such that $C_p = E$, only $EC3$ changes $C_p$. If $p$ executes $EC3$, then one of the following two conditions must be true: (i) $Par_p = \emptyset$ and $p$ is not a leaf, and (ii) $Par_p = \{r\}$ and $p$ is not a leaf of an illegal path. Both (i) and (ii) contradict our assumption.

2. In order to break a path $\mu_p = (p_1, p_2, ..., p_l)$ so that a live leaf is created, $\exists p_i \in \mu_p$ such that $p_i$ executes an action in $\alpha$ and $p_i$ becomes a live leaf in $\beta$. Since, $D_{p_i} \neq \perp$, $p_i$ can execute only $EC1$ and $EC2$ in $\alpha$.

   If $p_i$ executes $EC2$, then $C_{p_i}$ becomes equal to $E$. So, $p_i$ is not a live leaf.

   If $Enable(EC1, p_i, \alpha)$, then $C_{p_i} = E$ in $\alpha$ because $\sharp Par_{p_i} > 0$. Since the execution of $EC1$ does not change the color, $p_i$ cannot become a live leaf.

3. A processor, $p \neq r$, without a parent, cannot select a new descendant because both $Forward(p)$ and $Backward(p)$ are disabled at $p$.

We proved the contradiction in all three cases. □

We define $\mathcal{A}_2 \equiv \mathcal{A}_1 \wedge (LIL = 0)$.

We need the following lemma to prove $\mathcal{A}_1 \triangleright \mathcal{A}_2$.

**Lemma 4.11** *$\forall \alpha \vdash \mathcal{A}_1$, $\forall \beta$ such that $\alpha \mapsto \beta$, if the value of $LIL$ in $\beta$ is equal to that of $LIL$ in $\alpha$, then no live illegal leaf chooses (by Action $TC1$ or $TC2$) a descendant which is on an illegal path in $\alpha \mapsto \beta$.*

**Proof.** Assume that the leaf of an illegal root $\mu_p$ chooses a descendant which belongs to an illegal path $\mu_q$. If $\mu_q$ is a cycle, then $\mu_p$ becomes a rooted cycle. If $\mu_q$ is an illegal rooted path, then the leaf of $\mu_q$ becomes the leaf of both $\mu_p$ and $\mu_q$. In both cases, we found a contradiction since the value of $LIL$ in $\beta$ is less than that of $LIL$ in $\alpha$. □

**Theorem 4.12** $\mathcal{A}_1 \triangleright \mathcal{A}_2$.

**Proof.** By Lemma 4.10, LIL cannot increase. Assume that LIL eventually stays constant at $x > 0$. By Corollary 4.9 and Lemma 4.11, every illegal rooted path looses processors infinitely often by execution of $EC1$, and also gains processors infinitely often by execution of $TC1$ and $TC2$ by the leaf.

Let us call such a path a *worm.* There are two cases to consider.

1. Assume that eventually, the leaf of the worm chooses only the path-free processors. Then, it is clear that the worm eventually contains no $E$-colored processor (the path-free processors cannot be $E$-colored, as per the definition of Macro $UV_p$). Once the worm contains no $E$-colored processor, if the root $p$ of the worm executes $EC1$, then $(C_p \neq E) \wedge (D_p = q) \wedge (D_q = \perp) \wedge (C_q = C_p)$ (see Predicate $BrkD(p)$). In this case, after the execution of $EC1$, the worm disappears (both $p$ and $q$ become path-free) and $LIL$ decreases, which contradicts that $LIL$ never decreases.

2. Assume that the leaf of the worm infinitely often chooses processors which belong to a path. Let $q$ be such a processor. By Lemma 4.11, $q$ cannot be on an illegal path. Hence, $q$ belongs to the legal path $\mu_r$. Thus, the legal leaf (denoted by $\ell$ in the rest of the proof) is also the (illegal) leaf of the worm.

   Again by Lemma 4.11, while $\ell$ is the leaf of both $\mu_r$ and the worm, $\ell$ cannot choose a processor which is on an illegal rooted path. Thus, either $(a)$ $\ell$ eventually chooses a processor $q'$ in the legal path $\mu_r$ such that $q'$ is between $r$ and $q$, or $(b)$ $\ell$ chooses only the path-free processors (while $\ell$ is the leaf of both $\mu_r$ and the worm).

   a. $\ell$ eventually chooses a processor $q'$ in the legal path $\mu_r$. In that case, the worm becomes a rooted cycle. Then, $LIL$ decreases which contradicts that $LIL$ never decreases.

   b. $\ell$ chooses only the path-free processors. Then, by repeated execution of $TC1$ and $TC2$, $\ell$ eventually becomes the descendant of $q$ (as in a "normal" cround). Then, $q$ is enabled to execute $EC2$ followed by $EC1$ only. So, by fairness, $q$ eventually becomes the $E$-colored leaf of both $\mu_r$ and the worm. Thus, $LIL$ decreases which contradicts that $LIL$ never decreases.

□

**Corollary 4.13** *In any configuration* $\vdash \mathcal{A}_2$*, if there exists a live leaf, then it must be the legal leaf.*

## 4.4 Color Consistency

In this section, we show that eventually, either the system contains no live leaf, or every processor in the legal path (except the leaf) has the same color as $r$ has. In such a configuration, the legal path cannot create a new cycle. We first show (by using Theorem 4.6) that $r$ changes its color infinitely often. So, $r$ starts a new *cround* with a new color infinitely often. If the legal path, $\mu_r$ does not meet any illegal path, then it remains color consistent (all processors, except the leaf, have the same color). Otherwise, when $\mu_r$ meets an illegal path, its leaf becomes dead and it remains color consistent.

**Lemma 4.14** *The root* $r$ *changes its color infinitely often.*

**Proof.** By Theorem 4.6, $r$ executes an action infinitely often. $r$ can execute only $TC1$ and $TC2$. If $r$ executes $TC1$ infinitely often, then $r$ changes its color infinitely often and hence, the lemma is proven. Assume that $r$ does not execute $TC1$ infinitely often. This implies that $r$ executes $TC2$ infinitely often (by Theorem 4.6). Then, by the definition of $Search_r$, eventually $D_r = \bot$ must be true. This will enable $r$ to execute $TC1$, which contradicts our assumption. $\square$

We define a predicate *ColorConsistent* in a configuration $\gamma$ such that it is true if any of the following conditions is true: ($CC1$) $D_r = \bot$. ($CC2$) The leaf of the legal path is a live leaf, and all processors on the legal path, except the leaf (which may or may not have the same color as of $r$), are *r-colored* (meaning, they have the same color as that of $r$). ($CC3$) The legal path does not have a leaf, i.e., the path is a rooted cycle, or has a dead leaf.

We define $\mathcal{A}_3 \equiv \mathcal{A}_2 \wedge ColorConsistent$.

**Theorem 4.15** $\mathcal{A}_2 \triangleright \mathcal{A}_3$.

**Proof.** $\mathcal{A}_3$ is closed:

1. Assume that $D_r = \bot$. By Lemma 4.14, $r$ changes its color infinitely often. $r$ chooses a descendant by executing $TC1$. If $r$ chooses a descendant which belongs to a cycle or to an illegal rooted path with a dead leaf, then *ColorConsistent* remains true ($CC3$). If $r$ selects a path-free descendant $p$, then $p$ becomes the new live leaf of the legal path, $\mu_r$, and thus, *ColorConsistent* is preserved ($CC2$).

2. Assume that $D_r \neq \bot$. The only processor which can choose a descendant by executing $TC1$, is the live leaf of the legal path. Assume that $p$ is the live leaf. If $p$ chooses a path-free processor as the descendant (executing Action $TC1$ or $TC2$), then all processors except the leaf, are r-colored. Thus, *ColorConsistent* remains true ($CC2$). If $p$ selects a processor $q$ that is in a path, as the descendant, then the legal path ends in a cycle or a dead leaf (Theorem 4.12 and Corollary 4.13). Thus, *ColorConsistent* is preserved ($CC3$).

Every computation starting from a configuration satisfying $\mathcal{A}_2$ leads to a configuration in $\mathcal{A}_3$: The proof follows from Lemma 4.14. $\square$

### 4.5 Cycle Destruction

In this section, we prove that all cycles are eventually destroyed. The process of destruction is as follows: All strict cycles are merged with the legal path and thus, become rooted cycles. Then by the repeated application of $EC1$ and $EC2$, the rooted cycles will be destroyed.

We borrow the following term from [5] to simplify our presentation: The *first DFS tree* of the graph $G$ is defined as the DFS spanning tree rooted at $r$, created by traversing the graph in the DFS manner, and visiting the adjacent edges of every processor in the order induced by $\succ_p$. We defined the macro $Search_p$ such that Algorithm $\mathcal{TC}$ circulates the token in the first DFS tree.

**Lemma 4.16** *Starting from any configuration* $\vdash \mathcal{A}_3$*, every processor which belongs to neither the legal path nor any strict cycle, will eventually become path-free.*

**Proof.** $\forall \alpha \vdash \mathcal{A}_3$, every illegal path which is not a strict cycle, is a dead illegal rooted path. Any processor on an illegal rooted path can be enabled to execute Action $EC2$ once, followed eventually

by Action $EC1$ once. From Theorem 4.5, every processor $p$ on a dead illegal rooted path cannot be *Locked*. Thus, every $p$ eventually executes $EC1$. If $p$ is the root of the dead illegal rooted path, then $p$ becomes path-free and the lemma is verified. Otherwise ($p$ is not the root), after the execution of $EC1$, the descendant of $p$ becomes path-free. The repeated execution of $EC1$ along the path from the parent of the (dead) leaf towards the (illegal) root will make all processors on the illegal path path-free. □

**Lemma 4.17** *Starting from any configuration* $\vdash \mathcal{A}_3$*, every processor which is path-free and E-colored, will be eventually path-free and 0-colored.*

**Proof.** By fairness, all $E$-colored and path-free processors eventually execute $EC3$ because none of its neighbors can choose it as a descendant (in the macro $UV_p$, $q$ cannot be chosen if $C_q = E$ and $D_q = \perp$). □

**Lemma 4.18** *Starting from any configuration* $\vdash \mathcal{A}_3$*, every strict cycle will be eventually transformed into a rooted cycle.*

**Proof.** By Corollary 4.13, $\forall \alpha \vdash \mathcal{A}_3$, if there exists a live leaf, then it must be the legal leaf. By Lemma 4.14, the legal leaf becomes alive infinitely often. So, our obligation is to show that eventually, a processor in every strict cycle in the system will be selected as a descendant by the leaf of the legal path. Assume the contrary, i.e., there exists one strict cycle which will never be reached by the legal path.

In the rest of the proof, $SC$ will denote the first strict cycle never reached by the legal path which is on the first DFS tree. So, there exists $\alpha \vdash \mathcal{A}_3$ such that all processors between $r$ and $SC$ are path-free (by Lemma 4.16), or they belong to the legal path ($r$-colored in $\mathcal{A}_3$). By Lemma 4.17, $\forall e \in \mathcal{E}_\alpha :\ \exists \alpha' : \alpha \leadsto_e \alpha'$ such that every processor between $r$ and $SC$ is 0- or 1-colored. Also, by successive *crounds*, $\forall e \in \mathcal{E}_\alpha :\ \exists \alpha'' : \alpha' \leadsto_e \alpha''$ such that every processor between $r$ and $SC$ has the same color $k$ (0 or 1).

Let $q$ be the first processor in $SC$ that is on the first DFS tree. Let $p \in N_q$ be the parent of $q$ in the first DFS tree. Since $SC$ is not reachable by the legal path (by assumption), $C_q = k$. Otherwise, $p$ will eventually select $q$ as a descendant, which will contradict our assumption. But, in the next *cround*, $p$ will choose $q$ as a descendant because $C_p$ will be equal to $(k+1)\ mod\ 2$ (with $C_q = k$). Thus, we arrive at the contradiction. □

**Lemma 4.19** *Starting from any configuration* $\vdash \mathcal{A}_3$*, every cycle is destroyed.*

**Proof.** By Lemma 4.18, every strict cycle is eventually transformed into a rooted cycle. All rooted cycles are eventually destroyed by the repeated application of $EC1$ and $EC2$. □

Let $NC$ denote the number of cycles in the system.
We define $\mathcal{A}_4 \ \equiv\ \mathcal{A}_3 \wedge (NC = 0)$.

**Theorem 4.20** $\mathcal{A}_3 \triangleright \mathcal{A}_4$.

**Proof.** $\mathcal{A}_4$ is closed: By the definition of *Search*, *Forward*, and Action $TC1$, the legal leaf chooses a descendant which has a different color than its own. So, the legal leaf chooses only the path-free processors. Thus, no new cycle can be created in $\mathcal{A}_3$. Hence, $NC$ cannot increase.

Every computation starting from a configuration in $\mathcal{A}_3$ leads to a configuration in $\mathcal{A}_4$: Follows from Lemma 4.19. □

### 4.6 Legitimacy Predicate

In this section, we prove that the legitimacy predicate $\mathcal{L}_{\mathcal{TC}}$ eventually holds. Then we show that Protocol $\mathcal{TC}$ is self-stabilizing. From Corollary 4.13 follows:

**Lemma 4.21** $\forall \alpha \vdash \mathcal{A}_4 : \exists! p :: \; Token(p)$.

We define the following for a configuration $\gamma \vdash \mathcal{A}_4$:
$\mathcal{A}_5 \equiv \mathcal{A}_4 \wedge \mathcal{L}_{\mathcal{TC}}$.

**Theorem 4.22** $\mathcal{A}_4 \triangleright \mathcal{A}_5$.

**Proof.** $\mathcal{A}_5$ is closed: Follows from the definition of $\delta_0$ and Actions TC1 and TC2.
Every computation leads to $\mathcal{A}_5$: By Lemma 4.16, $\forall \alpha \vdash \mathcal{A}_4 : \; \forall e \in \mathcal{E}_\alpha : \; \exists \alpha' : \alpha \rightsquigarrow_e \alpha'$ such that all processors in the system are path-free or belong to the legal path. By Lemma 4.17, $\exists \alpha'' : \alpha' \rightsquigarrow_e \alpha''$ such that every path-free processor is 0- or 1-colored. Then by successive *crounds*, $\exists \beta : \alpha'' \rightsquigarrow_e \beta$ such that every processor is path-free and has the same color $k$ (0 or 1). If $k = 0$ in $\beta$, then the lemma is proven ($\beta = \delta_0$). If $k = 1$, then, in the next round, $k$ becomes equal to 0. □

**Lemma 4.23 (Closure and Convergence)** $\mathsf{true} \triangleright \mathcal{L}_{\mathcal{TC}}$.

**Proof.** Follows from Theorems 4.1, 4.12, 4.15, 4.20, and 4.22. □

**Lemma 4.24 (Correctness)** $\forall \alpha \vdash \mathcal{L}_{\mathcal{TC}} : \forall e \in \mathcal{E}_\alpha :: \; e \vdash \mathcal{SP}_{\mathcal{TC}}$.

**Proof.** In $\mathcal{L}_{\mathcal{TC}}$, only Actions TC1 and TC2 are executed. Thus, the token is passed among the processors in the depth-first search order during a *cround*, and any computation is a repetition of *crounds*. In any configuration of $\mathcal{L}_{\mathcal{TC}}$, only one processor may execute an action. So, only one processor has a token in any configuration $\vdash \mathcal{L}_{\mathcal{TC}}$. □

**Theorem 4.25 (Self-Stabilizing)** *Protocol $\mathcal{TC}$ is self-stabilizing.*

**Proof.** Follows from Lemmas 4.23 and 4.24. □

## 5 State Complexity

A processor $p$ in Algorithm $\mathcal{TC}$ uses two variables, $D_p$ and $C_p$. The variable $C_p$, for a processor $p \neq r$, can have 3 different values (0, 1, and $E$), whereas $C_r$ can have only 2 values (0 or 1). The variable $D_p$ can have $\Delta_p$ ($|N_p|$) plus one ($\perp$) values. So, a processor, $p \neq r$, needs to maintain $3 \times (\Delta_p + 1)$ states and $r$ needs $2 \times (\Delta_r + 1)$. Thus, the total number of configurations of the whole network is

$$2\,(\Delta_r + 1) \times \prod_{p \in V,\; p \neq r} 3\,(\Delta_p + 1)$$

It is worth mentioning here that all the previous papers computed the space complexity in terms of the number of *bits* only, not in terms of the *states*. We feel that the measurement in terms of the number of states is more accurate.

## 6 Concluding Remarks

We presented a self-stabilizing depth-first token circulation scheme on a general network with a distinguished root. Our algorithm can also be used to construct a DFS spanning tree simply by maintaining the parent pointers.

A solution to the problem of mutual exclusion in a network is to implement a token circulating from one processor to the next following some pattern. The token moves around the network. A processor having the token is granted access to the shared resource and can execute the code in the critical section.

Our solution to the depth-first token circulation problem can be used to solve the *mutual exclusion problem*. After stabilization, in Algorithm $\mathcal{TC}$, in each token circulation round, a processor $p$ holds the token as many times as its degree $\Delta_p$—once while satisfying $Forward(p)$ and $\Delta_p - 1$ times while $Backtrack(p)$ is true. Since the degree of the processors in the network is bounded, Algorithm $\mathcal{TC}$ implements a *strictly fair* token circulation (and mutual exclusion). By *strict fairness*, we mean that, while a processor $p$ is waiting for the token, any other processor $q$ ($q \neq p$) can get the token at most a bounded number of times: here $\Delta_q$ times, where $\Delta_q$ is the degree of $q$.

It is also easy to implement the *1-fair* mutual exclusion, i.e., in each token round, all processors will enjoy the critical section access *exactly once*. In this case, a processor $p$ can enter the critical section if and only if $Forward(p)$ is true.

The space requirement for a processor $p$ is $3(\Delta_p + 1)$ states (only $2(\Delta_p + 1)$ states for the root). The question of the optimal state requirement for this problem is still open.

## References


[1] J Beauquier and O Debas. An optimal self-stabilizing algorithm for mutual exclusion on bidirectional non uniform rings. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 17.1–17.13, 1995.

[2] GM Brown, MG Gouda, and CL Wu. Token systems that self-stabilize. *IEEE Transactions on Computers*, 38:845–852, 1989.

[3] JE Burns, MG Gouda, and RE Miller. On relaxing interleaving assumptions. In *Proceedings of the MCC Workshop on Self-Stabilizing Systems, MCC Technical Report No. STP-379-89*, 1989.

[4] JE Burns and J Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11:330–344, 1989.

[5] Z Collin and S Dolev. Self-stabilizing depth-first search. *Information Processing Letters*, 49:297–301, 1994.

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

[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. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[9] M Flatebo, AK Datta, and AA Schoone. Self-stabilizing multi-token rings. *Distributed Computing*, 8:133–142, 1994.

[10] S Ghosh. An alternative solution to a problem on self-stabilization. *ACM Transactions on Programming Languages and Systems*, 15:735–742, 1993.

[11] MG Gouda and FF Haddix. The stabilizing token ring in three bits. *Journal of Parallel and Distributed Computing*, 35:43–48, 1996.

[12] MG Gouda and N Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40:448–458, 1991.

[13] ST Huang and NS Chen. Self-stabilizing depth-first token circulation on networks. *Distributed Computing*, 7:61–66, 1993.

[14] C Johnen, G 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, 1997.

[15] 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, 1995.

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

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

[18] F Petit and V Villain. Color optimal self-stabilizing depth-first token circulation. In *I-SPAN'97, Third International Symposium on Parallel Architectures, Algorithms and Networks Proceedings*, pages 317–323. IEEE Computer Society Press, 1997.

[19] F Petit and V Villain. Color optimal self-stabilizing depth-first token circulation protocol for asynchronous message-passing systems. In *PDCS-97 10th International Conference on Parallel and Distributed Computing Systems Proceedings*, pages 227–233. International Society for Computers and Their Applications, 1997.

[20] F Petit and V Villain. A space-efficient and self-stabilizing depth-first token circulation protocol for asynchronous message-passing systems. In *Euro-par'97 Parallel Processing, Proceedings LNCS:1300*, pages 476–479. Springer-Verlag, 1997.

[21] F Petit and V Villain. Optimality and self-stabilization in rooted tree networks. *Parallel Processing Letters*, 1999. To appear.

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