A Fault-Tolerant Token Passing Algorithm on Tree Networks

Gianluigi Alari, Joffroy Beauquier, Ajoy K. Datta, Colette Johnen, Visalakshi Thiagarajan

Abstract— **We present a self-stabilizing token passing algorithm for a tree network. The algorithm is based on the 4-state mutual exclusion algorithm of Dijkstra [5] and works under the distributed daemon model of execution.**

Keywords— **Distributed algorithms, fault-tolerance, mutual exclusion, self-stabilization, token passing.**

I. Introduction

Dijkstra [5] pioneered the study of self-stabilization in distributed systems in 1974. A self-stabilizing system, regardless of the initial states of the processes and initial messages in the links, is guaranteed to converge to the intended behavior within finite time. Dijkstra introduced the property of self-stabilization in distributed systems and applied it to algorithms for mutual exclusion on a ring [5]. A solution to the mutual exclusion problem on a ring is to implement a token circulating from one process to the next in one direction; the token moves around the ring and a process having the token is granted access to the shared resource and may execute the code in the critical section; Brown, Gouda, and Wu take this view in [2]. Several self-stabilizing token passing algorithms for different topologies have been proposed in the literature [16]: Dijkstra [5], [6], [7], Burns and Pachl [3], Flatebo and Datta [9], and Flatebo, Datta, and Schoone [10] for a ring; Brown, Gouda, and Wu [2] and Ghosh [11] for a linear array of processors, and Tchuente [18] on general networks. Recently, Huang and Chen [13] presented a token circulation protocol for a connected network in non-deterministic 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. A memory efficient token passing protocol on general network is presented in [15]. This algorithm needs needs only $O(1)$ bits per edge of the network. In [5] Dijkstra presented a mutual exclusion algorithm on a ring of 4-state machines where the neighborhood relationship between the two exceptional machines is not exploited. Ghosh [11] proposed an alternate solution to the 4-state mutual exclusion problem originally presented in [5]; his algorithm is simpler in the sense that the behavior of both the exceptional machines is identical with respect to their sole neighbors and the behavior of a non-exceptional machine is symmetric with respect to both of its neighbors.

In this paper, we propose a token passing algorithm for tree networks that works under the distributed daemon execution model; our approach is based on the 4-state algorithm of Dijkstra [5] and its variation by Ghosh [11]. Although our algorithm relies upon an underlying tree network topology it is not less general than the protocol in [13] since a spanning tree of a network can be obtained by a number of self-stabilizing algorithms [1], [4], [12], [17]. Token passing on a spanning tree thus places no restriction on the topology of the underlying distributed system.

The remainder of the paper is organized as follows. In the next section, we introduce the system model and the formal definition of the problem. The token passing algorithm and the informal description of the algorithm are presented in Section III. Section IV provides the proof of correctness. Finally, we make our concluding remarks in Section VI.

II. Model, Notation, and Problem Definition

A. Model

We model a distributed system by an undirected, connected graph $G = (V, E)$ where V and E are the set of vertices (or nodes) and set of edges, respectively. The vertices represent the processes and links are denoted by the edges of the graph. Existing methods can be used to construct a rooted spanning tree $T = (V, E')$ of G with $E' \subseteq E$ in a self-stabilizing manner. Our algorithm then provides the token passing on the underlying tree topology. We will denote by $P_i : i \in \{1..n\}$ the set of system processes, where $|V| = n$, and by P_r the root process of T. Every process P_i in the system has a multiple-

G. Alari is with Unité d'Informatique, Université catholique de Louvain, Belgium. J. Beauquier and C. Johnen are with L.R.I./C.N.R.S., Université de Paris-Sud, France. A. Datta and V. Thiagarajan are with Department of Computer Science, University of Nevada Las Vegas.

reader/single-writer register r_i which is serializable with respect to read and write operations.

Our model supports composite atomicity so that a process can read the value of its own and its neighbors' registers and write its register in a single atomic step. The composite atomicity model is more restricted than the read/write model in which a single read or write operation constitutes an atomic step. However, by a simple modification according to the method described in [1] and [14], our protocol may be used in the more general read/write atomicity model.

B. The Problem

The problem of mutual exclusion is to ensure that at most one process can execute (or enter) the critical section at any time, and in a finite time, all processes get a chance to execute the critical section. network while ensuring that fairness is preserved.

In a distributed system, the global state can be defined as the vector of all local states. For purposes of self-stabilization, the set of all global states $\mathcal{G}$ is partitioned into two sets: the set $\mathcal{L}$ of *legitimate* states, and the remainder $\mathcal{I} = \mathcal{G} \backslash \mathcal{L}$, the set of *illegitimate* states. An algorithm is called *self-stabilizing* if the following two conditions hold: first, starting from an arbitrary global state, the system reaches a legitimate state within a finite amount of time; second, any step taken while it is in a legitimate state causes a transition to a legitimate state. The required properties for a self-stabilizing token passing algorithm are:

Token System *Token Uniqueness:* In a legitimate state, there exists no more than one token at any time.

Fairness or Starvation-Freedom: If a state is legitimate, then every process obtains the token within a finite amount of time, and may thus execute the critical section code.

Self-Stabilization *Stability or Closure:* If a state is legitimate, then after the the daemon schedules an action, the corresponding state transition results in a legitimate state.

Finite Convergence: Starting from any system state, the system will reach a legitimate state after a finite time.

III. Token Passing Algorithm on a Tree

The state of each process P_i is maintained by a read/write register r_i containing the following fields:

$r_i.Id$	the id of P_i
$r_i.Parent$	the id of the parent of P_i, except for the root, which has zero
$r_i.S$	the state of P_i
$r_i.FirstChild$	the first child of P_i
$r_i.CurrentChild$	the child P_i is considering

We assume that every process maintains a circular list of its children (in the tree $T = (V, E')$). We will denote the list of process P_i by $ChildrenList(i)$. A special element of this list is designated to be the first child of the process. Note that this list need not be stored at a process, and can be easily computed by a process P_i from the state of the neighbors and the total order of the process identifiers. $r_i.S$ is in {0,2} if P_i is the root of the tree, in {1,3} if P_i is a leaf, and in {0,1,2,3} otherwise. $r_i.Id$ is a non-corruptible constant. $r_i.Parent$ is maintained by the underlying self-stabilizing spanning tree protocol so that $r_r.Parent$ is *null*. Obviously, the leaf processes set $r_i.FirstChild$ and $r_i.CurrentChild$ to *null*.

In order to simplify the presentation of the rules, the following macros are used in the algorithm (all macros are defined for P_i):

NEXTCHILD(i)
{return $ChildrenList(r_i.CurrentChild).next$;}
LASTCHILD(i)
{if ($NEXTCHILD(i) = r_i.FirstChild$)
then return $TRUE$; else return $FALSE$; }
CURRENTCHILD(i)
{if ($r_{r_i.Parent}.CurrentChild = r_i.Id$)
then return $TRUE$; else return $FALSE$; }
The algorithm for process P_i is given below. [1]
{For the root process P_r}
R0 :: $(r_{r_r.CurrentChild}.S = r_r.S + 1) \wedge$
$\sim LASTCHILD(r)$
$\longrightarrow r_r.CurrentChild := NEXTCHILD(r)$
R1 :: $(r_{r_r.CurrentChild}.S = r_r.S + 1) \wedge LASTCHILD(r)$
$\longrightarrow r_r.S := r_r.S + 2;$
$r_r.CurrentChild := NEXTCHILD(r)$
{For the leaf process P_i}
R2 :: $(r_{r_i.Parent}.S = r_i.S + 1) \wedge CURRENTCHILD(i)$
$\longrightarrow r_i.S := r_i.S + 2$
{For the interior process P_i}
R3 :: $(r_{r_i.Parent}.S = r_i.S + 1) \wedge CURRENTCHILD(i)$
$\longrightarrow r_i.S := r_i.S + 1$
R4 :: $(r_{r_i.CurrentChild}.S = r_i.S + 1) \wedge$
$\sim LASTCHILD(i)$
$\longrightarrow r_i.CurrentChild := NEXTCHILD(i)$

[1] All + operations in the rules are modulo 4.

R5 :: $(r_{r_i.CurrentChild}.S = r_i.S + 1) \wedge LASTCHILD(i)$
$\longrightarrow r_i.S := r_i.S + 1$;
$r_i.CurrentChild := NEXTCHILD(i)$

We associate to rules R1, R2 and R3 the ability to hold the token and access the shared resource.

We now describe the system dynamics at stabilization, i.e., how the token circulates when the system is already in a legitimate state. The details of how the system stabilizes will be given in Section IV.

Suppose that the system is in a legitimate state $L \in \mathcal{L}$ such that the root has an even value of S (say x), all other processes have the same odd value of S ($x+1$), the root's $CurrentChild$ is its last child (say P_k), and every other process' $CurrentChild$ is its $FirstChild$. The current state is a legitimate state since the only process to enjoy a privilege (rule R1 is true) and to have the token is the root P_r; P_r may execute the critical section, then P_r will change $r_r.S$ to $x+2$, and advance its $CurrentChild$ pointer to its $FirstChild$ (say P_j). P_j now has the token (rule R3); it gets its turn to access the shared resource, and later moves changing $r_j.S$ to $x+2$. The token continues to traverse down the tree from the root following the $FirstChild$ pointers until it reaches a leaf process (say P_{m_1}); every process in the path from the root $FirstChild$ to P_{m_1}'s parent P_m got the token once (rule R3) and possibly entered the critical section. Suppose that all the children of P_m are leaf processes. P_{m_1} has the token and by rule R2 accesses the shared resource and changes its S to $x+3$. This amounts to backtracking the privilege (not the token) to its parent P_m that now moves by rule R4 passing the token to P_m's next child (say P_{m_2}) which gets its turn to enter the critical section (rule R2). The above process continues until P_m points to its last child (say P_{m_k}). P_{m_k} accesses the resource, increments its S value to $x+3$ by rule R2 passing again the privilege to P_m that now, by rule R5, advances its $CurrentChild$ to its $FirstChild$ P_{m_1} and changes its S to $x+3$ (equal to its last child's S). The case when not all children of P_m are leaf processes is analogous; it is sufficient to repeat the same arguments to each subtree rooted at P_m's non-leaf children.

Let us run the system until the last child of the root, P_k, moves by rule R5 or R2 setting $r_k.S$ to $x+3$ and pointing again to its $FirstChild$ and call this final global system state L'. It should be obvious now that starting from L, the token traverses the tree in depth-first-search (DFS) order. When the DFS completes, the system state L' is equivalent to the initial one: the root has $r_r.S = x+2$, all other processes have $S = x+3$; P_r points to its $LastChild$ P_k, and all other processes point to their $FirstChild$.

IV. Proof of Correctness

Definition 1: A privilege is a program guard whose corresponding action allows a process to modify the value of the "state" field of its read/write register.

Definition 2: A semi-privilege is a program guard whose corresponding action allows a process to modify the value of some of the register fields but not the "state" field.

The execution of an action due to a (semi) privilege is called a move and a process P_i is said to enjoy a (semi) privilege when at least one of its guards is satisfied and the process can make the corresponding move. We will denote by $P_Priv(i)$ a privilege depending on the value of S of the $Parent$ of process P_i (see guards of rules R2 and R3), by $C_Priv(i)$ a privilege depending on the value of S of the $LastChild$ of P_i (see guards of rules R1 and R5) and by $S_Priv(i)$ a semi-privilege depending on the value of S of the $CurrentChild$ (different from its $LastChild$) of P_i (see guards of rules R0 and R4).

Definition 3: A process P_i, $i \neq r$ holds a token if $P_Priv(i)$ is satisfied, and the root P_r holds a token if $C_Priv(r)$ is true.

Observation 1: A (semi) privilege is always related to a pair of processes (P_i, P_j) where $P_i = r_j.Parent$, $P_j = r_i.CurrentChild$, and $P_Priv(j) \rightarrow \sim (C_Priv(i) \vee S_Priv(i))$. Furthermore, if $|r_i.S - r_j.S| \ mod \ 2 = 1$, then one and only one among $C_Priv(i)$, $P_Priv(j)$, and $S_Priv(i)$ is satisfied.

Lemma 1: If P_i is such that $even(r_i.S)$, then there exists at least one process enjoying a (semi) privilege in the path from P_i to a leaf process following the CurrentChild pointers downwards.

Proof: The proof derives directly from Observation 1 and the fact that leaf processes have odd values of S. □

Lemma 2: There is at least one (semi) privilege in the system (no deadlock).

Proof: This follows directly from Lemma 1 and the fact that the root may only have an even value of S. □

Lemma 3: Every process must make a move in a finite amount of time.

Proof: Proof by contradiction.

Assume that process P_i with $r_i.S = x$ does not move. Then, the processes in the subtree T_i rooted at P_i may move only a finite number of times; otherwise, S of P_i's current child could reach the value $x+1$ and either $C_Priv(i)$ or $S_Priv(i)$ would hold.

Let x be even. Then by Lemma 1, there will always be at least a (semi) privilege in the subtree T_i contradicting the above hypothesis.

If x is odd, then the parent of P_i must move only a finite number of times; otherwise, it could reach the value $x-1$ and enable P_i when $CURRENTCHILD$ is satisfied. This argument must hold for all processes in the path from P_i up to the root process. This contradicts the fact that root, always having an even value of S, must make a move in a finite amount of time. □

Corollary 1: Eventually every process will enjoy a privilege and thus change its S field (no starvation).
Proof: Since by Lemma 3 every process must make a move in a finite amount of time, if P_i never changes $r_i.S$, it must always move due to a semi-privilege. This is a contradiction since, when $LASTCHILD$ holds, P_i may only move by Rules (R3) and (R5) and thus change $r_i.S$. □

We now define the set $\mathcal{L}$ of legitimate states and prove the closure of $\mathcal{L}$ and the finite convergence of our algorithm to a state $L \in \mathcal{L}$.

Being the set of global states $\mathcal{G}$ the product space of all local states, we define the set $\mathcal{L}$ of legitimate states for our token passing algorithm as follows:

Definition 4: Let $p(i)$ be the number of (semi) privileges enjoyed by P_i. $\mathcal{L} = \{g \in \mathcal{G} \mid \Sigma_{i=1}^{n} p(i) = 1\}$
This means that in a legitimate state there must be only one process enjoying only one (semi) privilege.

Observation 2: The only case in which a move by process P_i may generate two (semi) privileges is when $C_Priv(i)$ is satisfied and the following conditions are satisfied (see Figure 1 for an example): let P_p be P_i's parent, P_j and P_k be the first and last child of P_i, respectively; $|r_i.S - r_p.S|\ mod\ 2 = 0$, $|r_i.S - r_j.S|\ mod\ 2 = 0$ and $r_k.S = r_i.S + 1$. After P_i moves, its $CurrentChild$ is P_j and Observation 1 holds for the pair of process (P_i, P_j) and (P_p, P_i). In all other cases, a move may never produce more than one new (semi) privilege.

Observation 3: It follows from Observation 2 that if P_i is parent of leaf processes only, (i) it can never create two new privileges since its children are forced to have odd values of S, (ii) the state transition $odd(r_i.S) \rightarrow even(r_i.S)$ may only happen due to P_i's parent.

Lemma 4: The set $\mathcal{L}$ of legitimate states is closed under algorithm execution (stability or closure).
Proof: By Lemma 1, the system will always have at least one (semi) privileged process. This is sufficient to avoid a deadlock situation in which no (semi) privileged process exists. Thus, all that we need to prove is that the system, starting from a legal state in which there is a single (semi) privilege, will never have more than one (semi) privilege (Definition 4).

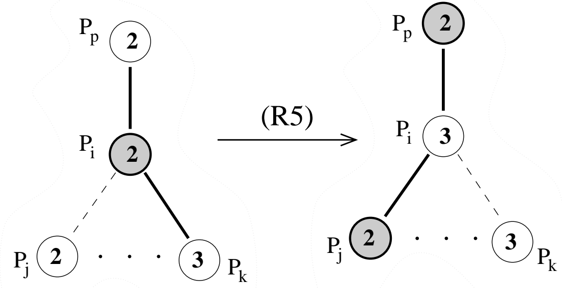

Fig. 1. Creating two new privileges in a single move: privileged processes are shaded and bold lines denote $CurrentChild$ pointers

Observation 2 shows the only situation where a move by a (semi) privileged process may generate two new (semi) privileges. This situation would take the system into an illegitimate state. Now, we need to prove that if a configuration, as described in the right side of Figure 1, is present in a global state, then this state cannot be a legitimate state.

From the configuration used in Observation 2, if $r_i.S$ is even, $r_j.S$ is also even. Then, by Observation 3, P_j is not a leaf process, and from Lemma 1, there exists at least one more (semi) privilege in the path from P_j to the leaf processes following the $CurrentChild$ pointers.

On the other hand, if $r_i.S$ is odd, then $r_k.S$ is even. As above, P_k is not a leaf and there exists at least one (semi) privilege in the path from P_k to the leaf processes. In both cases, there are at least two (semi) privileges and the system is not in a legitimate state. □

We will now prove that the convergence with respect to $\mathcal{L}$ is guaranteed by the algorithm.

Lemma 5: Consider a subtree T_i rooted at P_i such that P_i's k children $P_{i_1} \dots P_{i_k}$ are all leaf processes. After at most k moves of P_i, all processes in T_i will have odd values of S.
Proof: If $odd(r_i.S)$, then the Lemma is proved. Let $even(r_i.S)$ and $r_i.S = x$. Then by Corollary 1, P_i will change $r_i.S$ by Rules (R3) or (R5) in a finite amount of time and in both cases, $r_i.S$ will become odd. In the worst case, when $r_{i_j}.S = x - 1,\ j : 1..k$, and the $CurrentChild$ of P_i is P_{i_1}, P_i enjoys $k - 1$ semi-privileges which makes P_i point to its $LastChild$ P_{i_k}. Then P_i enjoys a privilege (due to

its parent or last child) to increment $r_i.S$ to $x+1$. □

Lemma 6: The system will eventually converge to a state where all processes but the root have an value of S state and thus there is only one privileged process (convergence).

Proof: Proof by induction on the height [2].

1. $height = 1$:

The system is composed only of the root and $n-1$ leaf processes and the lemma holds from Observation 1 and the fact that leaf processes have odd values of S and the root has an even value of S.

2. $height = l$:

Assume that the lemma holds for a tree of height at most $l-1 > 0$. Consider a subtree T_i as defined in Lemma 5 where P_i is at level $l-1$ and P_i's k children $P_{i_1} \ldots P_{i_k}$ at level l. Let P_j be the parent process of P_i and, without loss of generality, let all values of S values in T_i be odd and $r_i.S = x$. Then eventually, $P_Priv(i)$ will hold (Corollary 1), and P_i will move changing its state value from x to $r_j.S = x+1$ by rule (R3). By Corollary 1, P_j must move in a finite time and increment $r_j.S$ to $x+2$. P_j may only reach $x+2$ due to a privilege from its parent or a privilege from its last child. If P_j moved because of its last child, it implies that P_i set $r_i.S$ to $x+2$ and that, with respect to P_j, the subtree T_i is functionally equivalent to a single leaf process because the only way P_i may affect P_j is by the state transition $x \xrightarrow{x+1} x+2$ with x and $x+2$ being odd. We may then fuse all processes in T_i by only considering the single process P_i. This "fusion" process applies to all non-leaf processes at level $l-1$ and thus in a finite time we reduce the height of the tree by one. □

Theorem 1: The algorithm of Section III is a self-stabilizing algorithm with respect to the set of legitimate states $\mathcal{L}$.

Proof: It follows directly from Lemmas 4 and 6. □

Now we need to prove that the algorithm of Section III also satisfies the *Fairness* property. We will also show that our algorithm implements a strictly [3] fair token circulation scheme.

First, we show that eventually the system reaches a state where all processes but the root have the same odd value of S. Next we demonstrate how the token is passed in the network. Then our claim of strictly fair token passing will be obvious.

Definition 5: A legitimate state $S \in \mathcal{L}$ is a *Start* state if the following condition holds:

[2]The height of a tree is defined as the maximum level of the nodes of the tree. The level of a node is one plus the level of its parent, the level of the root being zero.

[3]A strictly fair token circulation scheme implies that every process gets the token equal number of times.

$LASTCHILD(r) = TRUE \wedge \forall i \neq r \mid ((r_i.S = r_r.S+1) \wedge (r_i.CurrentChild = r_i.FirstChild))$, i.e., the root has an even value of S (say x), all other processes have the same odd odd value of S ($x+1$), the root's $CurrentChild$ is its last child, and every other process' $CurrentChild$ is the process' $FirstChild$.

Lemma 7: Consider a subtree T_i rooted at process P_i such that P_i's k children $P_{i_1} \ldots P_{i_k}$ are all leaf processes. Starting from an arbitrary legitimate state, after at most $2(k-1)$ moves of P_i, all leaf processes in T_i will have the same odd value of S and P_i's $CurrentChild$ will be its $LastChild$.

Proof: Assume that all processes in T_i have odd values of S (Lemma 6). Since the system is in a legitimate state, from Corollary 1, in a finite time, $P_Priv(i)$ is satisfied and P_i makes a move setting $r_i.S$ to x such that $even(x)$. Next, again in a finite time, $C_Priv(i)$ is satisfied and P_i increments $r_i.S$ to $x+1$.

Let $P_{i_j} : i \leq j \leq k$ be the current child of P_i when $r_i.S$ becomes even and the only (semi) privilege of the system is confined in T_i. It is easy to verify that it takes exactly $k-j$ moves of P_i so that $r_i.CurrentChild = P_{i_k}$ (i.e., P_i points to its last child) and $r_{i_c}.S = x+1$, $c : j..k$. Assume that $j = 1$. Then P_i is in the state $even(x)$ and is pointing to its $FirstChild$, and the lemma is proved. Now assume that $j > 1$. First consider the case when P_i is not the root. P_i makes the $(k-j+1)^{th}$ move, sets $r_i.S$ to $x+1$, and points to its $FirstChild$ P_{i_1}. The next time $P_Priv(i)$ is satisfied, P_i goes through the state transition $x+1 \rightarrow x+2$, and then it makes exactly $k-1$ moves after which all its children have the value of S equal to $x+3$, and P_i's current child is its last child. Summing up, the total number of moves of P_i is $2k-j+2$ for $1 < j \leq k$, and k if $j = 1$. On the other hand, if P_i is the root, it always have an even value of S and the total number of moves is $2k-j$ for $1 < j \leq k$, and $k-1$ if $j = 1$. □

Corollary 2: Consider the subtree T_i as defined in Lemma 7 where $P_i \neq P_r$. Starting from an arbitrary legitimate state, after at most $2k+1$ moves of P_i, the system reaches a state where P_i has an even value of S ($=r_i.S$), all other processes in T_i have the same odd value of S ($=r_i.S+1$), and P_i's $CurrentChild$ is its $FirstChild$.

Proof: Follows from Lemma 7. □

Lemma 8: Starting from a legitimate state, the system will eventually converge to a *Start* state.

Proof: Proof by induction on the height.

1. $height = 1$:

The system is composed only of the root P_r and $n-1$ leaf processes $P_1 \ldots P_{n-1}$. By Lemma 7 in at

most $2(n-2)$ moves of P_r, all leaf processes will have the same odd value of S and the system will be in a *Start* state.
2. $height = l$:
Assume that the lemma holds for a tree of height at most $(l-1) > 0$. Consider a subtree T_i as defined in Lemma 7 where P_i is at level $l-1$ and P_i's children $P_1 \ldots P_k$ are at level l. By Corollary 2, since P_i is not the root, after at most $2k+1$ moves of P_i, all processes in T_i have the same odd value of S and P_i's $CurrentChild$ is its $FirstChild$. By using similar arguments as in Lemma 6, we may fuse all processes in T_i by considering the single process P_i thus reducing the height of the tree by one. □

The choice of associating a token to the rules $R1$, $R2$, and $R3$ only should be clear from the following observation:

Observation 4: When the system reaches a special *Start* state (Lemma 8), the token traverses the tree in the DFS order.

Theorem 2: The algorithm of Section III is a self-stabilizing token passing algorithm on a tree structured network. The token traverses the tree in the DFS order, hence implementing a strictly fair token management scheme.
Proof: Follows from Lemma 8 and Observation 4. □

V. Distributed Daemon

In this section we will show that the algorithm presented in Section III is such that the simultaneous moves of any subset of processes enjoying (semi) privileges can be serialized. So, the arguments used in proving closure and convergence in Section IV remain valid also for the distributed daemon model. Then we can claim that the algorithm of Section III solves the mutual exclusion problem on a tree structured system in the presence of a distributed daemon.

Definition 6: We define a privilege chain α as a list of length $|\alpha|$ of successive (semi) privileged processes $P_1 \cdots P_{|\alpha|}$ such that $P_{k+1} = r_k.CurrentChild$, $1 \leq k < |\alpha|$. A maximal privilege chain is a privilege chain such that the parent of the first process (if it exists) and the current child of the last process (if it exists) are not privileged.

Definition 7: We will say that a process P_i interferes with another process P_j if both P_i and P_j enjoy a (semi) privilege, and a move by P_i may cause P_j lose its (semi) privilege.

Remark 1: Processes executing in distinct maximal privilege chains cannot interfere with each other.

Let $\Sigma = \{P, C, S, R\}$ be a finite alphabet of symbols. It is possible to associate a privilege chain α to a finite string σ of symbols in Σ of length $|\alpha|$ such that the following conditions are true:
(i) P_i is the ith process in α.
(ii) $\sigma_i = P \rightarrow P_Priv(i)$.
(iii) $\sigma_i = C \rightarrow C_Priv(i)$.
(iv) $\sigma_i = S \rightarrow S_Priv(i)$.
(v) $\sigma_i = R \rightarrow (P_Priv(i) \wedge (S_Priv(i) \vee C_Priv(i)))$, i.e., P_i enjoys two (semi) privileges.

Lemma 9: Let α be a privilege chain and σ its corresponding string, then $\sigma = P^k.R^z.\{C,S\}^{l-k-z}$ where $l = |\alpha|$, $z \in \{0,1\}$, and $0 \leq k \leq l-j$.
Proof: The proof follows directly from the fact that if P_i and P_j are two successive processes in the privilege chain α such that P_j's parent is P_i, from Observation 1, if $P_Priv(j)$ holds, then $(C_Priv(i) \vee S_Priv(i))$ cannot hold. □

Lemma 10: The simultaneous execution of moves of the processes in a privilege chain α are serializable.
Proof: Let $\sigma = P^k.R^z.\{S,C\}^{l-k-j}$ correspond to a privilege chain α. Consider the following schedule of moves:
1. $z = 0$:
P_i moves before P_{i-1} for $1 < i \leq k$ and before P_{i+1} for $k < i < l$.
2. $z = 1$:
If the processes move simultaneously, if P_{k+1} moves according to rule (R3), then let P_i move before P_{i-1} for $1 < i \leq k+1$ and before P_{i+1} for $k+1 < i < l$. Otherwise, if P_{k+1} moves according to rule (R4) or rule (R5), let P_i move before P_{i-1} for $1 < i \leq k$ and before P_{i+1} for $k+1 \leq i < l$.

It is easy to verify that the above serial schedule of moves results in a final state that is the same as the one reached if all processes in α had moved simultaneously. □

Note that the above lemma holds for any privilege chain and in particular, for any subset of a maximal privilege chain.

Theorem 3: Any simultaneous execution of moves in the system is serializable. Thus the algorithm in Section III solves the mutual exclusion problem in the presence of a distributed daemon.
Proof: It follows directly from Lemma 10 and from the fact that simultaneous moves in different maximal privilege chains cannot interfere with each other. □

VI. Conclusions

In this paper, we presented an algorithm for self-stabilizing distributed mutual exclusion on a span-

ning tree under the composite atomicity model and distributed daemon hypothesis. We first proved its correctness under the more restrictive central daemon model of execution and then we extended the proof to the distributed daemon case by showing serializability of simultaneous moves.

The algorithm may be applied to distributed systems whose interconnection network is a connected graph by using layering techniques and combining it with one of the well known self-stabilizing spanning tree algorithms such as [1], [4].

The algorithm in [13] is a non-deterministic depth-first-search token circulation on a connected network whereas ours is a deterministic token circulation algorithm on tree networks. Furthermore, unlike [13], we formally prove the correctness of our algorithm under the distributed daemon execution model. With respect to [8], one limitation of our protocol is that it works under the distributed daemon with the composite atomicity model which is a stricter model than the read/write atomicity model used in [8]. However, Arora and Gouda in [1] and Huang, Wuu, and Tsai in [14] showed a simple method to transform a protocol designed for the composite atomicity model to an equivalent one working under read/write atomicity.

Both our protocol and the protocol of [8] proceed in phases or rounds starting with the token at the root process of the tree network and terminating when the token returns to the root process itself. However, the algorithm presented in this paper implements a strictly fair token circulation policy in which every process gets the token exactly once during a phase, whereas in [8], a process gets the token d times during a single round, where d is the degree of the node representing the process. Thus, if the resource needing mutual exclusion is critical for all processes in the system and if the degree of contention for this resource is high, then our protocol assures an equal distribution of accesses to all the system units.

Our algorithm requires $O(\log\ n)$ space because we need to store the Id, $FirstChild$, $CurrentChild$ and $Parent$ process identifiers; the variable S requires only *two bits*. The algorithm in [8] uses $O(\delta)$ memory where δ is the maximum degree of a node and Huang and Chen [13] have shown that their algorithm needs $O(n)$ memory.

If we implement our algorithm of Section III using read/write atomicity actions only, following the method in [1], we only need two additional 4-state variables at each process to maintain a copy of $Parent.S$ and $CurrentChild.S$. Thus, the memory requirement of our algorithm still remains $O(\log\ n)$.

References

[1] A. Arora and M. Gouda, "Distributed Reset," *Proceedings of the 10th Conference on Foundations of Software Technology and Theoretical Computer Science*, Bangalore, India, December 17-19, 1990, pp.316-331, Lecture Notes in Computer Science 472, Springer-Verlag; also *IEEE Transaction of Computers*, Vol. 19, No. 11, November 1993, pp. 1026-1038.

[2] G. Brown, M. Gouda, and M. Wu, "Token Systems that Self-Stabilize," *IEEE Transactions on Computers*, Vol. 38, No. 6, June 1989, pp. 845-852.

[3] Burns J. and Pachl J. "Uniform Self-Stabilizing Rings," *ACM Transactions on Programming Language and Systems*, Vol. 11, No. 2, 1989, pp. 330-344.

[4] N. Chen, H. Yu, and S. Huang, "A Self-Stabilizing Algorithm for Constructing Spanning Trees," *Information Processing Letters*, Vol. 39, 1991, pp. 147-151.

[5] E. W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control," *Communications of the ACM 17*, 1974, pp. 643-644.

[6] E. W. Dijkstra, "Self-Stabilization in Spite of Distributed Control," in *Selected writings on computing: a personal perspective*, Springer-Verlag, Berlin, 1982, pp. 41-46.

[7] E. W. Dijkstra, "A Belated Proof of Self-Stabilization," Distributed Computing, Vol. 1, No. 1, January 1986, pp. 5-6.

[8] S. Dolev, A. Israeli, and S. Moran, "Self-Stabilization of Dynamic Systems Assuming only Read/Write Atomicity," *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, Quebec City, Canada, pp. 103-117, 1990; also *Distributed Computing* Vol. 7, 1993, pp. 3-16.

[9] M. Flatebo and A. K. Datta, "Two-State Self-Stabilizing Algorithms for Token Rings," *IEEE Transactions on Software Engineering*, June 1994, pp. 500-504.

[10] M. Flatebo, A. K. Datta, and A. A. Schoone, "Self-Stabilizing Multi-Token Rings," *Distributed Computing*, Vol. 8, 1995, pp. 133-142.

[11] S. Ghosh, "An Alternate Solution to a Problem on Self-Stabilization," *ACM Transactions on Programming Languages and Systems*, Vol. 15, No. 4, September 1993, pp. 735-742.

[12] S. Huang and N. Chen, "A Self-Stabilizing Algorithm for Constructing Breadth First Trees," *Information Processing Letters*, Vol. 41, January 1992, pp. 109-117.

[13] S. Huang and N. Chen, "Self-Stabilizing Depth-First Token Circulation on Networks," *Distributed Computing*, Vol. 7, 1993, pp. 61-66.

[14] S. Huang, L. Wuu, and M. Tsai, "Distributed Execution Model for Self-Stabilizing Systems," *Proceedings of the 14th International Conference on Distributed Computing Systems*, Poland, 1994, pp. 432-439.

[15] C. Johnen and J. Beauquier, "Space-Efficient Distributed Self-Stabilizing Depth-First Token Circulation," *Proceedings of the 2nd Workshop on Self-Stabilizing Systems*, Las Vegas, Nevada, 1995, pp. 4.1-4.15.

[16] M. Schneider, "Self-Stabilization," *ACM Computing Surveys*, Vol. 25, No. 1, March 1993, pp. 45-67.

[17] S. Sur and P. K. Srimani,"A Self-Stabilizing Distributed Algorithm to Construct BFS Spanning Trees of a Symmetric Graph," *Parallel Processing Letters*, Vol.2, No. 2 & 3, September 1992, pp. 171-180.

[18] M. Tchuente, "Sur l'auto-stabilisation dans un reseau d'ordinateurs," *RAIRO Informatique Theorique 15*, No. 1, 1981, pp.47-66.