

The Self-Stabilizing Edge-Token and Its Applications*

SU-SHEN HUNG AND SHING-TSAAN HUANG[†]

*Department of Computer Science
National Tsing Hua University
Hsinchu, 300 Taiwan*

[‡]*Department of Computer Science and Information Engineering
National Central University
Taoyuan, 320 Taiwan*

Consider a connected graph with nodes (or processes) and edges (or communication links). An edge token associated with an edge is a token maintained by the two nodes connected by the edge; one and only one of the two nodes holds the token. An edge token can be passed from one node to the other if so desired. This paper first presents a randomized self-stabilizing algorithm to implement the edge token, in which each process maintains two three-state variables for an edge; the scheme works under the distributed scheduler with the read/write atomicity. Then, the edge token algorithm is used as a building block in two other self-stabilizing algorithms: one is for ring orientation problem and the other for token circulation problem on trees. All the proposed algorithms are uniform.

Keywords: edge token, leader election, mutual exclusion, orientation, self-stabilization

1. INTRODUCTION

The term *self-stabilization* was introduced by Dijkstra in 1974 [2]. A self-stabilizing system guarantees to converge to a legitimate state in a finite time no matter what initial state it may start with. This makes a self-stabilizing system be able to recover from transient faults automatically without any outside intervention.

A distributed system can be represented by a connected graph (V, E) , where V is the set of nodes (representing processes) and E is the set of edges (representing communication links). We say that nodes u, v are each other's neighbor if (u, v) is an edge in E . An *Edge Token* associated with an edge is maintained by the two nodes connected by the edge. One of the two nodes holds the token; the edge token may be passed from one node to the other. By using the edge token as a building block, we present two other self-stabilizing algorithms: one is for ring orientation; the other is for token circulation on trees.

Ring orientation concerns the agreement on a common orientation (direction) of the edges. There are many works in the literature on this problem. Due to the symmetry problem, there is no deterministic uniform self-stabilizing algorithm for rings of even size [12]. Papers in [5, 6, 20, 21] presented some deterministic self-stabilizing algorithms for odd size of rings with distributed daemons. Israeli and Jalfon [12] proposed a randomized

Received January 20, 2006; revised October 26, 2007; accepted May 29, 2008.

Communicated by Tsan-sheng Hsu.

* This research was supported in part by the National Science Council of Taiwan, R.O.C. under contracts No. NSC 91-2213-E008-011 and 92-2213-E008-029. The preliminary parts of the paper have been presented at the 6th Symposium on Self-Stabilizing Systems (SSS'03) and ISCA 17th International Conference on Parallel and Distributed Computing Systems (PDCS-2004).

algorithm for any size of rings with a distributed daemon. Later Hoover and Rudnicki [8] generalized the approach in [12] to orient edges in networks and considered the read/write atomicity issue. The read/write atomicity restricts that a process either reads the state of one of its neighbors or updates its local state but not both in an atomic step [3]. All of the previous works require $O(n^2)$ time for stabilizing, where n is the network size.

An algorithm for token circulation is usually to make a token circulate among the nodes of the network with each node getting the token at least once in every circulation cycle. In [15, 18, 19] Petit and Villain proposed depth-first token circulation algorithms for rooted tree networks. Subsequently, several protocols were proposed for general networks. In [3], the authors constructed a spanning tree first, then the token circulates on the spanning tree. Huang and Chen [9] presented a self-stabilizing depth-first token circulation algorithm for arbitrary rooted networks without constructing a spanning tree. Datta, Villain, Johnen and Petit also proposed depth-first token circulation algorithms for general networks with focus on space-efficiency [1, 13, 14, 16, 17]. All above algorithms need a distinguished root node; that is, they are not uniform. An algorithm is said to be *uniform* if all processes are anonymous and execute the same program. Most of the above algorithms need $O(n * D)$ stabilizing time except the algorithm in [16], whose stabilizing time is $O(n)$, where n is the network size and D is the degree of network.

In this paper, we first propose a uniform randomized self-stabilizing algorithm for the edge token. The paper in [7] provides the probability analysis for the randomized self-stabilizing algorithms. In the proposed algorithm, each process maintains only two three-state variables for each neighbor; the algorithm works under the distributed scheduler with the read/write atomicity. A similar three-state approach has been adopted in [12] for directing edges, but our design is different from the approach in [12]. In our design, edge tokens are used to identify the directions of edges (*i.e.* direction is from the process holding edge token to the other), however, in [12] tokens are fixed with constant directions. Besides that, we also extend our algorithm by providing two different simple but efficient token-passing mechanisms, and it can be applied to non-orientation algorithms.

Later, we propose a uniform self-stabilizing orientation algorithm for any size of rings and a uniform self-stabilizing token circulation algorithm for trees based on the edge token algorithm. The fair combination of self-stabilizing algorithms [3] is used to link the edge token algorithm with them.

In our orientation algorithm, each process only maintains 162 states. Meanwhile it only needs $O(n \log n)$ rounds to stabilize. Finally, we also present a uniform self-stabilizing token circulation algorithm for trees, again based on the edge token algorithm. It only needs $O(n)$ rounds to stabilize.

The rest of the paper is organized as follows. Section 2 describes the self-stabilizing edge token algorithm. The proposed ring orientation algorithm and its correctness proof are given in section 3. Section 4 presents the token circulation algorithm and its correctness proof. Finally, section 5 gives some concluding remarks.

2. THE SELF-STABILIZING EDGE TOKEN ALGORITHM

In this section, we propose a self-stabilizing algorithm to implement the edge token. In the following, we use *Etoken* for the term *edge token*. The algorithm guarantees that

after stabilizing, there is one and only one Etoken held by one of the two neighboring processes connected by the edge. The algorithm also provides two different mechanisms for passing the Etoken between the two processes.

Assume P_i, P_j are two neighboring processes. Let each process maintain one 3-state variable S ranging from 0 to 2. The S values are ordered with $0 < 1 < 2 < 0$. Also let S maintained by P_i be denoted as $S.i$, and S maintained by P_j be denoted as $S.j$. There are two cases are considered:

(1) $S.i \neq S.j$

The process with the larger S value (by the order $0 < 1 < 2 < 0$) holds the Etoken.

(2) $S.i = S.j$

We design a randomized rule to break the symmetry. A process randomly selects a new S value among 0 and 2. Once the symmetry is broken, there is one and only one Etoken held by P_i or P_j .

The algorithm also provides two Etoken passing mechanisms:

(1) Pushing Method

The process holding the Etoken has the privilege to decide whether to pass the Etoken to its neighbor or not. The privileged process just increases its S value by one when it wants to pass the Etoken. For example, assume $S.i (= 2) > S.j (= 1)$, then P_i holds the Etoken; P_i may pass the Etoken to P_j by setting $S.i = (S.i + 1) \bmod 3 = (2 + 1) \bmod 3 = 0$. Then, since $S.i (= 0) < S.j (= 1)$, P_j gets the Etoken.

(2) Pulling Method

The process not holding the Etoken has the privilege to decide whether to pull the Etoken from its neighbor or not. The privileged process just decreases its S value by one when it wants to pull the Etoken. For example, assume $S.i (= 2) > S.j (= 1)$, P_j does not hold the Etoken; it may pull the Etoken from P_i by setting $S.j = (S.j - 1) \bmod 3 = (1 - 1) \bmod 3 = 0$. Then, since $S.i (= 2) < S.j (= 0)$, P_j gets the Etoken.

One may adopt one of the two passing methods depending on the applications.

We also consider the read/write atomicity into our design. The read/write atomicity restricts that a process either reads the state of one of its neighbors or updates its local state, but not both in an atomic step [3]. In order to do this, we let each process maintain another 3-state variable that keeps the image of its neighbor's S . The image of $S.j$ maintained by P_i is denoted as $(S.j).i$ and the image of $S.i$ maintained by P_j is denoted as $(S.i).j$. The algorithm consists of two phases: the read-phase is to read the neighbor's S value to the image variable; the write-phase is to update the S value if needed. Each read-phase or write-phase is an atomic step. Under the distributed scheduler, many processes may be activated simultaneously, but theoretically we may serialize the activations and achieve the same result.

The algorithm for Etoken is shown as follows and the symbol $[]$ is used to bracket each atomic action.

Process P_i

Variables:

$S.i$: 0, 1, 2.

$(S.j).i$: image of $S.j$ at P_i ; P_j is a neighbor of P_i .

State predicate:

$\text{Token}(i, j) \equiv S.i > (S.j).i$ /* process i holds Etoken */

Procedure:

$\text{PassToken_Push}(i, j)$: **if** $\text{Token}(i, j)$ **then** $S.i := (S.i + 1) \bmod 3$ **fi** /* Pushing Method */

$\text{PassToken_Pull}(i, j)$: **if** $\text{Token}(j, i)$ **then** $S.i := (S.i - 1) \bmod 3$ **fi** /* Pulling Method */

Begin

Repeat forever {

[$(S.j).i = \text{read}(S.j)$] /* read-phase */

[**if** $(S.i = (S.j).i)$ **then** /* write-phase */

$S.i = \text{Random}(0, 1, 2)$

fi]

/* application area */

}

End

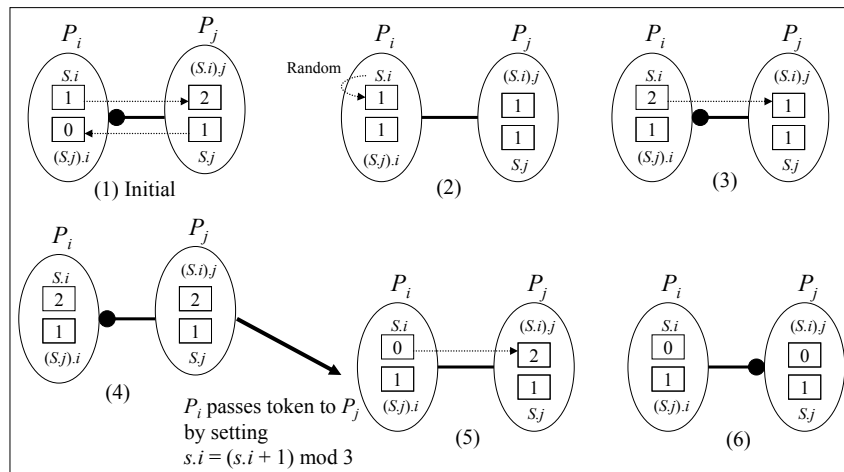


Fig. 1. An example for Etoken stabilizing.

Fig. 1 shows a stabilizing example for Etoken. In the figure the black circles indicate the edge tokens; the process closer to the Etoken holds the token. After both processes update the image variables, P_i randomizes its S value to break the symmetry at state (2). The Etoken is stabilized after state (4). Then states (5) and (6) show the Etoken passing by Pushing Method from P_i to P_j . Note that state (5) indicates a transient condition in which neither process holds the Etoken. However, this won't affect the correctness of the algorithm since P_i and P_j will do nothing before the image variable is updated to match its real value.

Next, we will prove the correctness of the algorithm.

Theorem 1 The Etoken algorithm is self-stabilizing.

Proof: Assume P_i, P_j are two neighboring processes. The stabilization predicate for Etoken(i, j) is $S_i \neq S_j$. In our design, randomization is applied when $S_i = S_j$. Once the symmetry is broken, the algorithm is stabilized. Since the probability to break the symmetry is $2/3$, the probability to stabilize is $\sum_{k=1}^{\infty} 1 - (1/3)^k = 1$. What remains to be argued is its correctness under the read/write atomicity. Since the only write operation in the algorithm is to randomize S values, so even when the image values are out of date, it may only cause the processes to apply randomization one more time and won't lead to any live-lock. Hence, the algorithm is self-stabilizing. \square

The algorithm can be easily generalized from two processes to multiple processes by letting a process maintain two 3-states variables for each of its neighbors.

For time complexity analysis, we adopt the measurement of round defined in [4].

Definition A *round* is the minimum time period that each process is scheduled to execute at least once.

Theorem 2 The expected time for all Etokens to stabilize is $O(\log n)$ rounds.

Proof: An Etoken is not stabilized if the S values of the two processes connected by the edge are the same. According to the algorithm, either one or both the processes can execute randomization. Assume there are k Etokens not stabilized at the beginning. Since each process can handle one Etoken at one time, among those k Etokens, we may assume that there are k_2 Etokens on which both processes take actions, k_1 Etokens on which only one process takes action, and k_0 Etokens on which no processes takes action. It is obviously that

$$k_0 + k_1 + k_2 = k. \quad (1)$$

There are totally $(2k_2 + k_1)$ processes taking actions on $(k_2 + k_1)$ Etokens; and $2k_0$ processes take no action on k_0 Etokens. Those $2k_0$ processes are among the $(2k_2 + k_1)$ processes. So

$$2k_0 \leq 2k_2 + k_1. \quad (2)$$

From Eqs. (1) and (2), we get:

$$\begin{aligned} 2k_0 &\leq 2k_2 + k_1 \leq 2(k_2 + k_1) = 2(k - k_0), \\ 4k_0 &\leq 2k, \\ k_0 &\leq k/2. \end{aligned} \quad (3)$$

The probability to break the symmetry is $2/3$ no matter one or two processes take actions. Therefore, after one round, the number of Etokens that remain not stabilized is

$$k - (2/3)(k_2 + k_1) = k - (2/3)(k - k_0) = (1/3)k + (2/3)k_0 \leq (1/3)k + (2/3)(k/2) = (2/3)k.$$

Hence, the expected time for all Etokens to stabilize is $O(\log n)$ rounds. \square

Moreover, the Etoken algorithm can be used as a building block (“slave” protocol) to combine with other self-stabilizing algorithms (“master” protocols) by the fair combination technique in [3] to achieve more jobs. According to the definition of fair combination, assume that Pr_2 is self-stabilizing for a task T_2 given task T_1 . If Pr_1 is self-stabilizing for T_1 , then the fair combination of Pr_1 and Pr_2 is self-stabilizing for T_2 . Here, we assume that the master protocols only use the Etoken passing mechanisms that provided by Etoken algorithm. We will prove that the combined algorithm is still self-stabilizing.

Theorem 3 Assume protocol Pr is self-stabilizing for a task T given the Etoken passing mechanisms, then the fair combination of Pr and Etoken algorithm is self-stabilizing for T .

Proof: According to Theorem 1, the Etoken algorithm is self-stabilizing. Once Etoken algorithm is self-stabilizing, the executing of Etoken passing procedure does not change the legitimate state, that is $S.i \neq S.j$ for process i and j . The closure property of Etoken algorithm is satisfied. Hence, according to the definition, the fair combination of Pr and Etoken algorithm is self-stabilizing for task T . \square

Next, we will focus on two “master” protocols: one is ring orientation algorithm; the other is token circulation algorithm.

3. THE RING ORIENTATION ALGORITHM

3.1 The Proposed Algorithm

In this section, we propose a self-stabilizing orientation algorithm for uniform rings. The algorithm is based on the Etoken algorithm presented in the previous section and adopts the Pushing Method in passing Etokens.

For a ring of n processes, there are n edges; hence there are n Etokens. The idea is to make each process hold exactly one Etoken. As in Fig. 2, assume that the direction of an edge points from the process holding the Etoken to the other process; then obviously, the ring is oriented if each process holds exactly one Etoken.

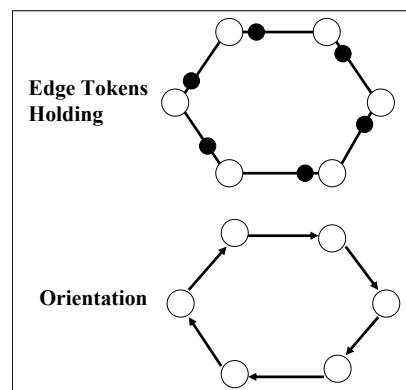


Fig. 2. The orientation of rings by using Etoken.

When a process holds both Etokens of its two edges, it keeps one and passes the other. The problem is that the Etoken may be bounced back and forth if the Etoken passing does not follow control. To solve this problem, we introduce another 2-state variable D . Let $D.i$ points to the last token received. That is, when there is one token and a new one is received, let $D.i$ points to the new one.

A formal description of the algorithm is as follows:

Process P_i

Variables:

$D.i$: a two-state pointer.

Begin

/* Etoken algorithm is as described in section 1 */

if (P_i holds one Etoken) **then**

Let $D.i$ point to the Etoken that P_i holds.

fi

elseif (P_i holds two Etokens) **then**

PassToken_Push($i, D.i$); /* P_i push the Etoken which $D.i$ points */

Let $D.i$ point to the Etoken that P_i keeps

fi

End

3.2 The Correctness Proof and Analysis

In this section, we prove the correctness of the ring orientation algorithm and analyze its space and time complexities.

3.2.1 Correctness proof

Definition Let X denote the number of processes holding no Etoken in the ring.

Here we will prove that the algorithm stabilizes to $X = 0$ eventually, that is, each process holds one and only one Etoken.

Lemma 1 X is non-increasing.

Proof: This is a direct consequence from the fact that only the processes holding two Etokens would pass one of them out. Therefore, once a process holds an Etoken, it will always keep one at least. \square

Lemma 2 (Liveness) If the ring is not oriented, at least one process has the privilege to take actions.

Proof: If the ring doesn't stabilize, there must exist some process holding no Etoken and some process holding two Etokens. The process holding two Etokens has the privilege to take actions. \square

Definition A segment is a maximum sequence of neighboring processes holding Eto-

kens with one and only one process holding two Etokens at one end. A segment with k processes holds exactly $(k + 1)$ Etokens.

Fig. 3 shows an example for segments (the gray parts indicate segments). In this example, there are three segments (k, a, b) , (d, e) and (g, h, i) . The ring is separated into segments by the processes holding no Etoken, such as the processes c, f and j in this example. Note also that the number of segments is equal to the number of processes holding no Etoken. By the definition, there is no segment when the ring is oriented.

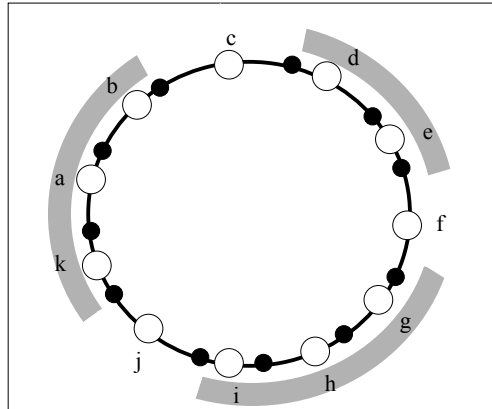


Fig. 3. An example of segments.

Lemma 3 It takes at most a round for the D pointers to be stabilized.

Proof: Due to arbitrary initialization, D pointers may not be correct. According to the algorithm, it is obvious that it takes at most a round for the D pointers to be stabilized. \square

As mentioned before, a segment with k processes holds exactly $(k + 1)$ Etokens. Consider the passing of the extra Etoken over the segment. By using the D pointer, the process always pushes out the old Etoken when it receives a new one. Hence, the extra Etoken moves in a constant direction, and eventually reaches a node with no Etoken outside the segment. Note that the node outside the segment may get an Etoken from the next segment before the mentioned extra token reaches it.

Lemma 4 X monotonically decreases as long as $X > 0$.

Proof: When $X > 0$, there must exist at least one segment. According to Lemma 3, the D pointers in the segment will stabilize, and then by Lemma 2, the extra Etoken of the segment moves in a constant direction, and eventually reaches a node with no Etoken outside the segment. That is, the node with no Etoken eventually gets an Etoken; and hence, X decreases. If the node already got an Etoken before the mentioned extra Etoken reaches it, then X decreased already. Hence, by Lemma 1, this lemma holds. \square

Theorem 4 The algorithm stabilizes to $X = 0$ eventually.

Proof: This is a directly consequence of Lemma 4. \square

3.2.2 Complexity analysis

In this subsection, we analyze the space complexity and time complexity of the proposed algorithm. In the analysis, n denotes the ring size.

Theorem 5 In the algorithm, each process only needs 162-state memory space.

Proof: In our algorithm, each process only maintains two 3-state variables for each of the two neighbors, and another 2-state variable; that is, each process only needs memory space of $(3 \times 3)^2 \times 2 = 162$ states. \square

Lemma 5 After all Etokens and D pointers stabilized, it takes at most n rounds to decrease X by half.

Proof: Consider the passing of the extra Etoken over the processes of a segment. It takes a round to pass the Etoken from a process to its neighbor. Since the maximum size of a segment could be n , it takes at most n rounds for the extra Etoken to reach a node outside the segment. Note that the node outside the segment may get an Etoken from the next segment before the mentioned extra Etoken reaches it. That is, due to paralleled execution, two segments can reduce X at least by one in n rounds. That is, it takes at most n rounds to decrease X by half. \square

Lemma 6 The maximum possible X is $n/2$.

Proof: In the worst case, there are $n/2$ segments in the ring. That is, there are at most $n/2$ processes without Etoken. Therefore, the maximum possible X is $n/2$. \square

Theorem 6 The expected stabilizing time of the algorithm is $O(n \log n)$ rounds.

Proof: According to Theorem 2, Lemmas 3, 5 and 6, the expected stabilizing time of the algorithm is $(\log n + 1 + n * \log(n/2))$ rounds, or $O(n \log n)$ rounds. \square

4. THE TOKEN CIRCULATION ALGORITHM

4.1 The Proposed Algorithm

In this section, we propose a self-stabilizing token circulation algorithm for uniform trees. The algorithm is also based on the Etoken algorithm, but adopts Pulling Method to pass the Etokens.

For a tree of n processes, there are $n - 1$ edges; hence there are $n - 1$ Etokens. It implies that there is at least one process holding no Etoken. The idea is to let the process holding no Etoken own the privilege, viz. hold the system token (Stoken). The privileged process releases the Stoken by getting an Etoken from one of its neighbors. If the neighbor holds no Etoken after that, the Stoken is passed to the neighbor; otherwise the Stoken just

disappears. Hence, when the Stokens traverse the entire tree, some of them will be disappeared during the process. The proposed algorithm guarantees there is one and only one Stoken eventually and the Stoken circulates among the processes of the tree fairly.

Similar to the ring orientation algorithm, the Etoken may be bounced back and forth between two nodes if the Etoken passing does not follow a fair pattern. To solve this problem, we use another variable D , which helps to decide which Etoken should be pulled from when a node has the Stoken. More specifically, $D.i$ is a pointer ranging from 1 to $N.i$, where $N.i$ denotes the degree of the process. Let $D.i$ points to the Etoken that it should pull back next time when it has the Stoken. Besides that, D variable also assures a fair token circulation. That is, Stoken will traverse the entire tree and each process has the fair opportunity to get the Stoken after self-stabilizing.

A formal description of the algorithm is as follows:

Process P_i

Variables:

$D.i$: a pointer pointing to one Etoken.

$N.i$: the degree of process i .

Begin

/* Etoken algorithm is as described in section 1 */

if (P_i holds no Etoken) **then** {

/* Enter the critical section */

PassToken_Pull($i, D.i$); /* Pull the Etoken from the neighbor that $D.i$ points to */

$D.i := (D.i + 1) \bmod N.i$

}

fi

End

4.2 The Correctness Proof and Analysis for the Token Circulation Algorithm

In this section, we prove the correctness of the token circulation algorithm and analyze its time complexity.

4.2.1 Correctness proof

According to our design, the process holding no Etoken owns the Stoken. In the following, we will prove that there exists one and only one Stoken in the tree eventually, and the Stoken fairly circulates among the processes of the tree. Before proceeding, let us make the following definitions first:

Definition A process holding no Etoken is called **s -process**.

Definition A process holding one and only one Etoken is called **e -process**.

Definition A process holding more than one Etokens is called **m -process**.

Definition Let $\#S$ denote the number of Stokens in the tree, that is, the number of s -processes in the tree.

Lemma 7 $\#S$ is always greater than zero.

Proof: There are $n - 1$ Etokens for a tree of n processes, hence there must exist at least one process holding no Etoken. That is, there exists at least one Stoken in the tree. Hence, $\#S$ is always greater than zero. \square

Lemma 8 $\#S$ is non-increasing.

Proof: According to the algorithm, only s -process can take action. When s -process takes action, it pulls an Etoken either from (1) a neighboring m -process or (2) a neighboring e -process. For case (1), $\#S$ decreases. For case (2), $\#S$ doesn't change. Therefore $\#S$ is non-increasing. \square

Lemma 9 $\#S$ monotonically decreases as long as $\#S > 1$.

Proof: $\#S > 1$ implies that there exists some m -processes. Since the existing Stokens are fairly circulating according to the usage of the D pointer, some Stoken shall hit the m -process and cause the Stoken to disappear. Therefore $\#S$ monotonically decreases as long as $\#S > 1$. \square

Theorem 7 The algorithm stabilizes to $\#S = 1$ eventually.

Proof: This is a direct consequence of Lemmas 7, 8 and 9. \square

Theorem 8 After the algorithm stabilizes, the Stoken circulates among the tree fairly.

Proof: This is obvious according to the usage of the D pointer. \square

4.2.2 Complexity analysis

In this subsection, we analyze the time complexity of the token circulation algorithm.

Definition An s -tree is a sub-tree that consists of one and only one s -process and maximum possible e -processes connected with the s -process via e -processes. And let s_i -tree denote the s -tree with process i being the s -process.

Lemma 10 No process belongs to two different s -trees.

Proof: The proof is by contradiction. Assume process x belongs to s_s -tree and s_t -tree. It implies that there exists a path $(p_s, \dots, p_x, \dots, p_t)$ starting from p_s to p_t through p_x . Assume the path consists of m processes ($m > 2$). By the definition of s -tree, the processes on the path not including p_s and p_t are all e -processes. That is, it only contains $(m - 2)$ Etokens. This contradicts the fact that there must exist $(m - 1)$ Etokens because there are $(m - 1)$ edges on the path. \square

Lemma 11 If $\#S > 1$, s -trees must be separated by m -processes.

Proof: This is a direct consequence of Lemma 10 and the definition of s -tree. \square

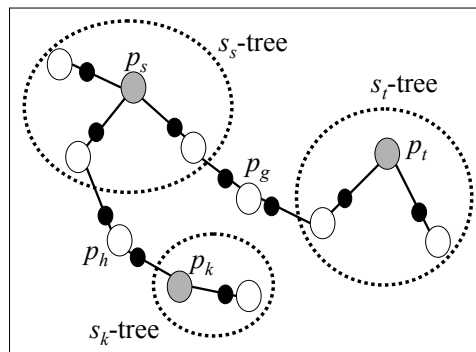


Fig. 4. There are three s -trees: s_s -tree, s_t -tree and s_k -tree separated by m -processes P_h and P_g .

Fig. 4 shows an example for s -trees. In the example, there are three s -trees. They are separated by m -processes P_h and P_g .

Lemma 12 It takes $2(k - 1)$ rounds for a surviving Stoken to traverse all the processes in an s -tree of k processes.

Proof: By the usage of the D pointer, it is obvious that each edge is traversed twice for a token circulation; hence it takes $2(k - 1)$ rounds for an Stoken to traverse all the processes in an s -tree of k processes provided that the Stoken does not disappear. \square

Theorem 9 After edge tokens stabilize, the token circulation algorithm takes at most $2(n - 1)$ rounds to stabilize, where n denotes the tree size.

Proof: Since an s -tree contains at most n processes, according to Lemma 12, an Stoken takes at most $2(n - 1)$ rounds to traverse all the processes in an s -tree. Before the circulation complete, the Stoken must hit an m -process and disappear; otherwise there is no m -process (i.e., $\#S$ decreases to one already). Hence the token circulation algorithm takes at most $2(n - 1)$ rounds to stabilize. \square

According to Theorems 2 and 9, the expected time for the token circulation algorithm to stabilize is $(\log n + 2(n - 1))$ rounds, or $O(n)$ rounds.

5. CONCLUDING REMARKS

In this paper, we present a randomized self-stabilizing algorithm to implement the edge token. In the algorithm, a process maintains two three-state variables for each edge; the scheme works under the distributed scheduler with the read/write atomicity. Then, the edge token algorithm is applied to solve two fundamental problems by appropriate combination in distributed systems: the orientation problem for rings and the token circulation problem for trees. Randomization is adopted to break the symmetry in the edge token algorithm. Although the other two algorithms are basically deterministic, they are also randomized algorithms after combining with edge token algorithm. In the edge token

algorithm, a process is allowed to read the state of its neighbor; however, for the two other algorithms, each process only refers to its own variables and take actions accordingly.

For the ring orientation algorithm, each process maintains 162 states. The maximum stabilizing time of the algorithm is $O(n \log n)$ rounds. For the token circulation algorithm, the time complexity is $O(n)$ rounds. Both results are better than previous works either in their time complexity or simplicity.

ACKNOWLEDGEMENT

The authors want to express their sincere gratitude to all the anonymous reviewers for the invaluable comments and suggestions provided in the reviews of the preliminary versions of the paper.

REFERENCES

1. A. K. Datta, C. Johnen, F. Petit, and V. Villain, "Self-stabilizing depth-first token circulation in arbitrary rooted networks," *Distributed Computing*, Vol. 13, 2000, pp. 207-218.
2. E. W. Dijkstra, "Self stabilizing systems in spite of distributed control," *Communications of the Association of the Computing Machinery*, Vol. 17, 1974, pp. 643-644.
3. S. Dolev, A. Israeli, and S. Moran, "Self-stabilizing of dynamic systems assuming only read/write atomicity," *Distributed Computing*, Vol. 7, 1993, pp. 3-16.
4. S. Dolev, A. Israeli, and S. Moran, "Uniform dynamic self-stabilizing leader election," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, 1997, pp. 424-440.
5. J. H. Hoepman, "Uniform deterministic self-stabilizing ring orientation on odd-length rings," in *Proceedings of the 8th Workshop on Distributed Algorithms*, 1994, pp. 265-279.
6. J. H. Hoepman, "Self-stabilizing ring orientation using constant space," *Information and Computation*, Vol. 144, 1998, pp. 18-39.
7. T. Herman, "Probabilistic self-stabilization," *Information Processing Letters*, Vol. 35, 1990, pp. 63-67.
8. H. J. Hoover and P. Rudnicki, "Uniform self-stabilizing orientation of unicyclic networks under read/write atomicity," *Chicago Journal of Theoretical Computer Science*, Vol. 5, 1996, pp. 1-37.
9. S. T. Huang and N. S. Chen, "Self-stabilizing depth-first token circulation on networks," *Distributed Computing*, Vol. 7, 1993, pp. 61-66.
10. S. T. Huang and S. S. Hung, "Self-stabilizing token circulation on uniform trees," in *Proceedings of the 6th International Symposium on Self-Stabilizing Systems*, 2003, pp. 92-101.
11. S. T. Huang and S. S. Hung, "Self-stabilizing edge-token and its applications," in *Proceedings of the 17th International Conference on Parallel and Distributed Computing Systems*, 2004, pp. 315-320.
12. A. Israeli and M. Jalfon, "Uniform self-stabilizing ring orientation," *Information and Computation*, Vol. 104, 1993, pp. 175-196.

13. C. Johnen, G. Alari, J. Beauquier, and A. K. Datta, "Self-stabilizing depth-first token passing on rooted networks," in *Proceedings of the 11th International Workshop on Distributed Algorithms*, LNCS 1320, 1997, pp. 260-274.
14. C. Johnen, J. Beauquier, and O. Debas, "Space-efficient distributed self-stabilizing depth-first token circulation," in *Proceedings of the 2nd Workshop on Self-Stabilizing Systems*, 1995, pp. 4.1-4.15.
15. F. Petit, "Highly space-efficient self-stabilizing depth-first token circulation for trees," in *Proceedings of International Conference on Principles of Distributed System*, 1997, pp. 221-235.
16. F. Petit, "Fast self-stabilizing depth-first token circulation," in *Proceedings of the 5th International Workshop on Self-Stabilizing Systems*, 2001, pp. 200-215.
17. F. Petit and V. Villain, "Color optimal self-stabilizing depth-first token circulation," in *Proceedings of the 3rd International Symposium on Parallel Architectures, Algorithms and Networks*, 1997, pp. 317-323.
18. F. Petit and V. Villain, "Time and space optimality of distributed depth-first token circulation algorithms," in *Proceedings of DIMACS Workshop on Distributed Data and Structures*, 1999, pp. 91-106.
19. F. Petit and V. Villain, "Optimality and self-stabilization in rooted tree networks," *Parallel Processing Letters*, Vol. 10, 2000, pp. 3-14.
20. M. S. Tsai and S. T. Huang, "Self-stabilizing ring orientation protocols," in *Proceedings of the 2nd Workshop on Self-Stabilizing System*, 1995, pp. 16.1-16.14.
21. N. Umemoto, H. Kakugawa, and M. Yamashita, "A self-stabilizing ring orientation algorithm with a smaller number of processor states," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, 1998, pp. 579-584.



Su-Shen Hung (洪淑慎) received her Ph.D. degree in 2005 from the Department of Computer Science, National Tsing Hua University, Taiwan. She had worked for the Information and Communications Research Laboratories, Industrial Technology Research Institute, Taiwan during 1989-2007 focusing on system software development and network planning & management. Her research interests include computer networks and distributed systems.



Shing-Tsaan Huang (黃興燦) was born in Taiwan on September 4, 1949. He got his Ph.D. degree in 1985 from the Department of Computer Science, University of Maryland at College Park. Dr. Huang is currently a professor at the Department of Computer Science and Information Engineering, National Central University, Taiwan. He is a K. T. Lee chair professor of the University and an IEEE fellow. His research interests include operating systems and distributed computing.