A Self-Stabilizing Algorithm for the Generalization of the Mutual Exclusion Problem

Mehmet Hakan Karaata and Rachid Hadid

***Abstract*—In this paper, we present the first stabilizing solution to the ℓ-exclusion problem in arbitrary networks. The ℓ-exclusion problem is a generalization of the mutual exclusion problem to ℓ ($\ell \geq 1$) processes, instead of 1, are free to use a shared resource simultaneously. The algorithm is semi-uniform and its space requirement is $(\ell + 3)\Delta r$ states for the root r, $4 \times \Delta 2p \times L_{max}$ states for each non root process p, where Δp is the degree of process p and L_{max} is the diameter of the communication network. This is the first ℓ-exclusion algorithm on arbitrary networks with the property that the space requirement is independent of ℓ for all processes except the root. The proposed protocol is distributed, deterministic, and does not use a pre-constructed spanning tree. Since our algorithm is self-stabilizing, it does not require initialization and withstands transient faults. The stabilization time of the algorithm is $O(\lceil n/l \rceil \times (\ell + L_{max}))$ rounds.**

***Index Terms*—Distributed systems, fault-tolerance, self-stabilization, ℓ-exclusion, propagation of information with feedback.**

I. Introduction

In 1974, Dijkstra introduced the property of self-stabilization in distributed systems and applied it to algorithms for mutual exclusion [1]. Self-stabilizing algorithms are able to withstand transient failures. We view a fault that perturbs the state of the system but not the program as a transient fault.

The ℓ-exclusion problem is a generalization of the mutual exclusion problem where ℓ processes are allowed to execute the critical section concurrently. A number of ℓ-exclusion algorithms are available in the literature [2]. The problem was first defined and solved by Fischer, Lynch, Burns, and Borodin in a generalized test and set model. The first token-based self-stabilizing algorithm for the ℓ-exclusion problem was presented in [3]. This solution is a generalization of Dijkstra's algorithm [1] on a ring. The algorithm in [4] is the second solution to the ℓ-exclusion problem, but unlike the first solution it assumes the shared memory model. In both cases the space requirement depends on the size of the network and ℓ. Algorithms in [4] and [3] require $O(2n)$ and $\Omega(nl)$ states per process, respectively, where n is the size of the network. Algorithm [5] works on tree and requires $O(Max^{2\Delta+1})$ where Δ is degree of the network and $Max \geq \ell$. First attempts to solve the ℓ-exclusion problem with a space complexity independent of n (and almost independent of ℓ) are presented in [6] for rings, and [7] for trees.

The algorithm in [8], present the first self-stabilizing ℓ-exclusion in message passing model in tree networks. In [9], two randomized uniform solution in unidirectional rings are presented. The first algorithm requires $(\ell \times \log(n))$ states per processor and the second algorithm requires $(\ell \times \log^2(n))$ states per processor. Recently, [10] propose a random walk solution in message passing model in ad hoc network. The drawback of this kind of the solution is that the waiting time of processor to enter the critical section is not bounded.

In this paper, we present the first self-stabilizing ℓ-exclusion algorithm in arbitrary networks. We provide an extension of the approach introduced in [7] to arbitrary networks. Our algorithm is token-based: a process can enter its critical section only upon receipt of a token. Our algorithm uses the well-known propagation of information with feedback scheme. Specifically, we use the new PIF scheme, called Propagation of information with Feedback and Cleaning (PFC) introduced in [11]. In our algorithm, all processes distribute tokens in the breadth-first manner, i.e., tokens are passed to different neighbors (provided more than one neighbor exists) following a local ordering. The space requirement of our algorithm is $(\ell + 3)\Delta r$ states (or $\lceil \log((\ell + 3)\Delta r) \rceil$ bits) for the root r, $4 \times L_{max} \times \Delta 2p$ states (or $\lceil 4 \times \log(2 \times L_{max} \times \Delta p) \rceil$ bits) for non root process p, where Δp is the degree of process p and L_{max} is the diameter of the network. This is the first algorithm on arbitrary networks in which the state space requirement is independent of the size of the network and l, except for the root. The stabilization time is $O(\lceil n/l \rceil \times (L_{max} + \ell))$ rounds.

The rest of the paper is organized as follows: in Section II, we describe the distributed system, the model we use in this paper, and also, state the specification of the problem solved in this paper. Then we present the proposed algorithm in Section III. The proof is omitted due to space constraints. Finally, we make some concluding remarks in Section IV.

II. Preliminaries

We consider a distributed system as an undirected connected graph $G = (V, E)$, where V is a set of nodes ($|V| = n$) and E the set of edges. Nodes of G represent processes and edges represent bidirectional communication links. We assume that processes and communication links are anonymous i.e., they do not have identifiers. We consider networks which are asynchronous and rooted, i.e., among the processes we distinguish a particular process called root and noted r. A communication link (p, q) exists if and only if p

Manuscript received November 25, 2012; revised March 13, 2013.

Mehmet Hakan Karaata is with Department of Computer Science and Department of Computer Eng P.O. Box 5969, Safat 13060 Kuwait (e-mail: karaata@gmail.com).

Rachid Hadid is with MIS, Universite´ de Picardie Jules Verne , 33, rue Saint Leu, 80039 Amiens Cedex 01, France.

and q are neighbors. For convenience, we assume that each process p labels its links 1, 2, ..., Δp and the labels of p are locally ordered by $\prec_p$. To simplify the presentation, we refer to the link from p to q (where q is one of the neighbors of p) at p by simply q.

In our computation model, each process executes the same program except the root. The distributed program of any process consists of a set of locally shared variables (henceforth referred to as variables) and a finite set of actions. A process can only write to its own variables, and read its own variables and that owned by the neighboring processes. Each action is of the following form: < label >:: < guard > → < 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 one or more variables of p.

An action can be executed only if its guard evaluates to true. If a guard is true, then the corresponding action is said to enabled, and disabled otherwise. A process is called enabled if it has at least one action enabled. The state of a process is defined by the values of its variables. The state of a system is a product of the states of all processes ($\in V$). In the sequel, we refer to the state of a process and system as a (local) state and configuration, respectively. Let C, the set of all possible configuration of the system. Let a distributed protocol P be a collection of binary transition relations denoted by $\mapsto$, on C. A computation of a protocol P is a maximal sequence of configurations $e = (\gamma_0, \gamma_1, ..., \gamma_i, \gamma_{i+1}, ...)$, such that for $i \geq 0$, $\gamma_i \mapsto \gamma_{i+1}$ (called step) if γ_{i+1} exists, else γ_i is a terminal configuration. All computations considered in this paper are assumed to be maximal. The set of all possible computations of P in a system is denoted E. We assume that each synchronous computation step, all actions enabled at the start of the step are executed concurrently by a concurrent distributed daemon. In synchronous round all processes read their neighboring states to determine the enabled guards, and then all these actions are executed concurrently before the next synchronous step.

1) Privilege: The definition of privilege to enter the critical section is the same as in [7], [12]: A processor has the privilege "if and only if it is enabled to make a particular move". The privileged action has the mark PR. In this paper, a processor is privileged if it holds a token.
2) Specification of the ℓ-exclusion protocol.: We consider a computation e of P to satisfy SPP of the protocol if the following conditions are true:

- Safety. In any computation e, at most ℓ processes can execute their critical sections concurrently.
- Fairness. In any computation e, each requesting process can enter the critical section in a finite time.
- Liveness. In any computation e, if $k < \ell$ processes execute the critical section forever and some other processes are requesting the critical section, then eventually at least another process will eventually enter the critical section.

An ℓ-exclusion algorithm is self-stabilizing if every computation starting from an arbitrary initial configuration, eventually satisfies the above safety, liveness, and fairness requirements.

III. Self-Stabilizing ℓ-Exclusion Algorthim

In this section, we present a stabilizing ℓ-exclusion algorithm where $\Delta r > \ell$ (other case is a simple version of this algorithm).

The Algorithm is shown in III.1 for the root and other processes.

A. Basis of the algorithm

The proposed algorithm works in two concurrent phases: The token distribution phase where the root starts a wave to distribute ℓ tokens down the network and the PIF synchronization phase which cleans the trace of the token distribution phase so that the root is subsequently ready to initiate a new token distribution phase.

These two phases are launched by the root alternately and carried out concurrently. After the root distributes all ℓ tokens then it can starts the PIF phase to clean up the network from the distributed tokens. The clean up phase (PIF) follows but not meets the token distribution and the token distribution phase terminates before the PIF phase. The goal of the token distribution phase is twofold: first, l tokens are distributed to the network using a wave in a fair manner and second, a spanning tree rooted at the root of the network is constructed. The token distribution phase is initiated by the root process by distributing ℓ tokens to its neighbors one at a time in order. If some tokens remain after distributing a token to each neighboring process, the root continues to distribute tokens again starting from the first neighbor. This is repeated until all the tokens are distributed. Upon receipt of a token, each process assumes the sender to be its parent and forwards the token to the next neighboring process (based on its local order of neighbors) that has not been included in the tree if such process exists. To determine the next neighboring process, each process keeps track of where it sent the last token using a pointer that circularly advances after sending each token. If a process fails to find a neighbor that has not been included in the tree and it does not have a child then the process destroys its token and becomes a leaf in the tree. On the other hand, if it fails to find a neighbor that has not been included in the tree, but has one or more children, it sends the token to its next child. The root always sends its token to the next process based on its local order of neighbors even if this process is already a child of a non-root process. So, this process accepts the token from the root and consequently becomes its child. Since during the token distribution every process records the process from which it receives a token as its parent so a tree rooted at the root is gradually built. But, we should note that many token distribution phases may be necessary to complete the construction of the tree since every process joins the tree after receiving a token and one phase may be not sufficient to reach all the processes of the system. However, after the construction of the complete spanning tree, each process distributes its tokens only among its children.

After all ℓ tokens are distributed, the PIF Propagation of Information with Feedback (PIF) synchronization phase is initiated by the root process to clean up the network between two consecutive token distributed phases. The PIF is carried

out only on the spanning tree built by the token distributed phases in a top-down manner. Since only those processes who have forwarded their tokens to their children are allowed to participate in this phase, PIF propagation follows and does not affect the token distribution phase. Once the leaves of the tree are reached by the PIF, they initiate the feedback and their parents relay the feedback to the root. When root receives the feedback from all its children, then it knows that the tree is cleaned up from the previous tokens and then starts a new token distribution phase. The repetition of the token distribution in this manner ensures that every process will receive a token (fairness) and the complete spanning tree of the network is eventually built. The safety is clearly obtained from the fact that the token distribution starts only after cleaning the tree from the preceding tokens and the root distributes exactly l tokens in the tree.

The self-stabilizing ℓ-exclusion algorithm is shown in Algorithm III.1. The token distribution phase is implemented using the privileged actions a_1, a_2, and a_6 and are marked as "(PR)". So p is privileged or has a token iff one of the following conditions holds:

1) p is the root (p=r), and a_1 or a_2 is enabled.
2) p is non-root processor and a_6 is enabled.

The cleanup process (PIF) is implemented using actions a_4 and a_5 for the root, a_7, a_8, and a_9 for other processes. Before describing the two phases of our algorithm in detail, we first introduce the variables maintained by each process p.

- T_p is used to implement both the token passing mechanism and the PIF synchronization. $T_p \in \{0, ...l-1, B, C, R\}$ for the root process and $T_p \in \{T\ ok, B, F, C\}$ for a non-root process.
- S_p denotes the neighbor to which p sent its last token.
- P_p denotes the parent of p. If there exists a neighbor of q ($q \in N_p$) such that $P_p = q$, then q is said to be the parent of process p and p is said to be a child of q. If a process p is not a parent of any process then process p is said to be a leaf or terminus process. Otherwise, p is said to be an internal process. Since the root never receives any token from any of its neighbors, it does not need to maintain P_p. So, we show this variable as a constant in the root's algorithm.
- L_p denotes the length of the path followed by the token from the root p. Again, since the root never receives any token from any of its neighbors, L_r must be 0, and hence, is shown as a constant in the algorithm

3) Token Distribution: As explained above, during the token distribution phase, the root process sends a wave containing l tokens to its neighbors and each token sent follows a path from root r until it reaches the terminus of the path (a leaf process) where it disappears. Moreover, a spanning tree rooted at r is built during the token passing process. A switch mechanism is used during the token distribution to ensure that every processor gets a token infinitely often. The switch mechanism is maintained at every process and implemented using a macro (not a variable but a dynamically evaluated function) Next_p to identify, using the pointer variable S_p, the next neighbor to be visited by the token. For any process p, Next_p returns the id of its next neighbor that has not been included in the tree, if exists. Otherwise, it returns the id of the next child among its ordered set of children, if exists, otherwise, i.e., p does not have neighbor not included in the tree nor a child, p destroys the token since its a leaf. However, before process p identifies its parent, multiple processes (called potential parents) may simultaneously send their tokens to p. Then, among all these potential parents, p chooses the root process to receive the token from if the root is also a potential parent of p; otherwise, p chooses the neighboring process with the smallest link number (Macros P_{arp} and P otentialp).

The T variable of the root process T_r is in state C before participating in the next token distribution phase. Subsequently, the root uses the successive values 0, ..., $\ell - 1$ of the variable T_r to differentiate the distribution of its ℓ tokens. A non-root process q receives a token when its T_q variable is in state C and one of its neighbors p such that T_p = T ok (or $\in \{0, ..., \ell - 1\}$ if p is root) holds, p has selected q as its potential child by assigning q to its S_p variable and the T variable of next process to receive its token, if any, is equal to C. When a leaf process assigns T_{ok} to its T variable, token propagation ends and the trace of this token propagation (values in T variables) are cleaned by the following PIF wave.

Whenever the root or an internal processor p receives a token, it selects the next neighbor (say q) to receive the token by advancing its pointer variable S_p to q. The token is passed by the root to one of its neighbors by executing either action a_1 (for the first token) or a_2 (for the second through the ℓ-th token). The token is received by a non-root process by executing a_6.

When a process q discovers that its parent or one or many neighboring potential parents are sending their tokens to it and if q has not yet a parent, then q is involved in this phase as follows (a_6):

- If q has no parent, then it chooses its parent p by assigning the link number associated to p to its variable Pq (consequently, q becomes a child of p). This leads process q to join the tree rooted at r.
- decides its level by assigning $Lp + 1$ to its variable Lq ,
- selects the next process (if any) by advancing Sq to the next neighbor (using $Next_q$) in its ordered sequence of neighbors to determine the recipient of the new token. If q fails to find a neighbor to transmits its token to, then it destroys the token and becomes leaf process in the tree, and
- q passes its token by changing its T value to T_{ok}.

When p uses the token by executing a_6, p cleans the trace of this token (T_{ok} value) with a C value (action a_9). Then p becomes ready either to receive another token of the same cycle or to execute the next phase (PIF synchronization phase). The root cleans the trace of this token (for the 0 to l − 2 tokens) with the next token number (a_2) or a R value (for the ($\ell - 1$) th token) (a_3).

4) PIF Synchronization.: After root sends its ℓ-th token and before starting the distribution of a fresh wave of ℓ tokens, it must be sure that the tree built during the preceding phases is cleaned up from tokens of the previous wave, i.e., all the distributed tokens are

consumed and disappeared at the leaves. This is done by setting the T variable of every process in the tree to a *C* (Cleaning) value. The clean up process is implemented using the PIF scheme. To implement this phase we specifically use the PFC (Propagation of Information with Feedback and cleaning) introduced in [11]. For that purpose, we need to use some additional values and variables. So, $T_p = B$ and $T_p = F$ refers to the broadcast and feedback state, respectively.

The root uses another additional values *R* of T_r to represent the ready to synchronize state. The root is in the ready to synchronize state before it initiates the PFC. After root sends its last token (the ℓ-th token), it sets its *T* variable to *R* (Ready to synchronize) to indicate to its children to be ready for a new PFC (action a_3). Subsequently, all its children alter their *T* variables to *C* (action a_9). Then, root starts a new PFC by switching T_r variable to B (action a_4). When process $p \in$ V -{r} with $T_p = C$ discovers that its parent process with $T = B$ then p participates in the broadcast phase and changes its T_p to *B* (action a_7). When the broadcast phase reaches a leaf process, the leaf process knows that all its ancestors entered the broadcast phase and starts the feedback phase by assigning F to its *T* variable (action a_8). Then, upon finding all its children in state F, each internal process p participates in the feedback phase by assigning *F* to its *T* variable (also action a_8). Consequently, the feedback phase propagates towards the root in a bottom-up manner and eventually reach root *r*. Every process *p* in the tree initiates the cleaning phase by setting its T_p value to *C* when each of its children and its parent q is either in the feedback phase ($T_q = F$) or in the cleaning phase ($T_q = C$) (action a_9). The purpose of the cleaning phase is to clean the trace of the preceding PFC phase. The cleaning phase works in parallel and pursues the feedback phase. Once all the children of the root enter the feedback phase, root participates in the cleaning phase (action a_5) causing the system to enter in the next phase of the algorithm and start a new ℓ-tokens distribution. Thus, the PFC wave works in parallel and follows the token distribution phase. The PFC wave should not be allowed to meet any token, i.e., the PFC wave cannot interfere with the token distribution phase. We implement this constraint as follows: A process p can change T_p to *B* only if T_p has a value C and all of its children have the *C* value, and its parent P_p has the value *B* (see action a_7).

B. Error Correction.

During the normal behavior, all system processes must preserve some properties based on the value of their variables and those of their parents. For each non-root process p the following properties need to be maintained.

1) For each process p which has already chosen its parent (i.e., $P_p = q$), the following properties need to be maintained.
- The parent *q* of p has also chosen its parent i.e., $P_q \neq \perp$.
- The distance L_p of process p is one plus that of the parent i.e., $L_p = L_q + 1$.
- If a process *p* is in the broadcast phase, then its parent *q* is also in the broadcast phase.
- If a process *p* is in the feedback phase, then its parent *q* is either in the broadcast, cleaning, or feedback phase.
- If a process p is in a token distribution phase (i.e., $T_p = T_{ok}$), then its parent *q* is either in the token distribution phase, cleaning phase, or ready to synchronize phase if q is the root process; token distributed phase or cleaning phase if *q* is non-root process.

2) For each process p which has not yet chosen its parent (i.e., $P_p = \perp$), then p is in a cleaning phase.

Algorithm III.1 Self-Stabilizing ℓ-Exclusion Algorithm

Input N_p: set of (locally) ordered neighbors;
$L_{max} = n - 1$;

Constants For the root $L_p = 0$; $P_p = \perp$;

Variables For the root $T_p \in \{0, 1, ..., \ell - 1, R, B, C\}$; $S_p \in \{1, ..., \Delta_p\}$;
For the non root $T_p \in \{Tok, C, B, F\}$; S_p, $P_p : \{1, ..., \Delta_p\}$;
$L_p : 0, ..., L_{max}$;

Macro

$$Potential_p = \begin{cases} q \in N_p :: T_q \in \{Tok, 0, 1, ..., \ell - 1\} \wedge S_q = p \\ \wedge\, P_q \neq p \wedge L_q < L_{Max}\} \end{cases}$$

$$Child_p = \{q \in N_p :: P_q = p\}$$

$$PotChild_p = \{q \in N_p :: P_q = \perp\}$$

$$Par_p = \begin{cases} r & \text{if } (r \in Potential_p) \\ P_p & \text{if } (P_p \in Potential_p) \\ min_{\prec_p}(Potential_p) & \text{otherwise} \end{cases}$$

$$Next_p = \begin{cases} min_{\prec_{S_p}}(N_p) & \text{if } (p = r) \\ min_{\prec_{S_p}}(Child_p \cup PotChild_p) & \text{if } (p \neq r) \\ \quad \wedge\, (Child_p \cup PotChild_p \neq \emptyset) & \\ \perp & \text{Otherwise} \end{cases}$$

Actions

For the root node

Predicates

$1stToken(p) \equiv (T_p = C) \wedge (\forall q \in N_p :: T_q = C)$

$Tth Token(p) \equiv (T_p \in \{0, ..., \ell - 2\}) \wedge (S_p = q \Rightarrow T_q = Tok) \wedge (T_{Next_n} = C)$

$Ready\text{-}To\text{-}Broadcast(p) \equiv (T_p = \ell - 1) \wedge (\forall q \in N_p :: T_q \in \{Tok, C\}) \wedge (S_p = q \Rightarrow T_q = Tok)$

$Broadcast(p) \equiv (T_p = R) \wedge (\forall q \in N_p :: T_q = C)$

$Cleaning(p) \equiv (T_p = B) \wedge (\forall q \in N_q :: T_q = F)$

/* Token distribution and Cleaning */

$(a_1) :: 1stToken(p) \rightarrow S_p := Next_p$; $T_p := 0$; **(PR)**

$(a_2) :: Tth Token(p) \rightarrow S_p := Next_p$; $T_p := T_p + 1$; **(PR)**

$(a_3) :: Ready\text{-}To\text{-}Broadcast(p) \rightarrow T_p := R$;

/* PIF Synchronization */

$(a_4) :: Broadcast(p) \rightarrow T_p := B$;

$(a_5) :: Cleaning(p) \rightarrow T_r := C$;

For other nodes

Predicates

$Normal(p) \equiv ((P_p = q) \Rightarrow (P_q \neq \perp) \wedge (L_p = L_q + 1) \wedge (T_p = B \Rightarrow T_q = B) \wedge (T_p = F \Rightarrow T_q \in \{B, F, C\}) \wedge (T_p = Tok \Rightarrow T_q \in \{Tok, 0, 1, ..., \ell - 1, R, C\})) \wedge ((P_p = \perp) \Rightarrow (T_p = C))$

$Token(p) \equiv (T_p = C) \wedge Normal(p) \wedge (Potential_p \neq \emptyset) \wedge (Next_p = q \Rightarrow T_q = C)$

$Broadcast(p) \equiv (P_p \neq \perp) \wedge (T_p = C) \wedge Normal(p) \wedge (T_{P_p} = B) \wedge (\forall q \in Child_p :: T_q = C)$

$Feedback(p) \equiv (T_p = B) \wedge Normal(p) \wedge (\forall q \in Child_p :: Child_q = F)$

$Cleaning(p) \equiv Normal(p) \wedge (T_p = F \Rightarrow T_{P_p} \in \{F, C\}) \wedge (T_p = Tok \Rightarrow ((S_{P_p} = p \Rightarrow T_{P_p} \in \{R, C\}) \wedge (S_p = q \Rightarrow (T_q = Tok \vee L_p \doteq L_{max}))))$

$AbnPath(p) \equiv \neg Normal(p) \wedge (P_p = q \Rightarrow (P_q = \perp) \vee (L_p \neq L_q + 1))$

$AbnPif(p) \equiv \neg Normal(p) \wedge (P_p = q \Rightarrow (P_q \neq \perp) \wedge (L_p = L_q + 1))$

/* Token distribution*/

$(a_6) :: Token(p) \rightarrow S_p := Next_p$; $T_p := Tok$; $P_p := Par_p$; $L_p := L_{P_p} + 1$; **(PR)**

/* PIF Synchronization and Token Clean */

$(a_7) :: Broadcast(p) \rightarrow T_p := B$;

$(a_8) :: Feedback(p) \rightarrow T_p := F$;

$(a_9) :: Cleaning(p) \rightarrow T_p := C$;

/*Correction action */

$(a_{10}) :: AbnPath(p) \rightarrow P_p := \perp$;

$(a_{11}) :: AbnPif(p) \rightarrow T_p := C$;

A process conforming to the above conditions is said to be in a normal state (Predicate Normal (*p*)). Otherwise, it is said to be in an abnormal state. For satisfying these properties, the correction actions a_{10} and a_{11} (Algorithm III.1) are used.

IV. Conclusion

In this paper, we presented the first stabilizing ℓ-exclusion

algorithm in arbitrary networks. This algorithm uses the PIF scheme and the Breadth-First token distribution. This makes our approach quite different than that followed by any other ℓ-exclusion algorithm. Our algorithm stabilizes in only $O(\lceil n/l \rceil \times (L_{max} + \ell))$ rounds. Its space requirement is $(\ell + 3)\Delta r$ states (or $\lceil \log((\ell + 3)\Delta r) \rceil$bits) for the root r, $4 \times L_{max} \times \Delta 2p$ states (or $\lceil 4 \log(2 \times L_{max} \times \Delta p) \rceil$bits) for non root process p. This is the first algorithm on arbitrary network in which space requirement is independent of ℓ for any process except one. A drawback of our algorithm, as in many deterministic self-stabilizing solutions to this problem in the current literature ([4], [7]), we cannot ensure that every execution of our algorithm always satisfies the liveness property: some processes may have to wait for others which are in their critical section, even if the total number of processes in the critical section is less than ℓ. Precisely, our algorithm allows at most one token to exist in a sequence of three processes. However, based on the assumption $\ell \leq \lceil n \rceil$, 3 we can observe that in any computation on numerous tree topologies, there exist some configurations where ℓ processes hold a privilege concurrently. Implementing a solution which satisfies the liveness property is a future challenge.

REFERENCES

[1] E. W. Dijkstra, "Self stabilizing systems in spite of distributed control," *Communications of the Association of the Computing Machinery*, vol. 17, no. 11, pp. 643–644, 1974.

[2] Y. Afek, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "A bounded first-in, first-enabled solution to the ℓ-exclusion problem," in *Proc. the 4th International Workshop on distributed Algorithm*, *LNCS*, Springer-Verlag, vol. 486, 1990, pp. 422-431.

[3] M. Flatebo, A. K. Datta, and A. A. Schoone, "Self-stabilizing multi-token rings," *Distributed Computing*, vol. 8, pp. 133–142, 1994.

[4] U. Abraham, S. Dolev, T. Herman, and I. Koll, "Self-Stabilizing ℓ-exclusion," *TCS, Theoretical Computer Science*, vol. 266, no. 1-2, pp. 653-692, 2001.

[5] G. Antonoiu and P. K. Srimani, "Self-stabilizing depth-first multi-token circulation in tree networks," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 16, no. 1, pp. 17–35, 2000.

[6] V. Villain, "A key tool for optimality in the state model," in *DIMACS'99, The 2nd Workshop on Distributed Data and Structures*, Carleton University Press, pp. 133–148, 1999.

[7] R. Hadid, "Space and time efficient self-stabilizing ℓ-exclusion in tree networks," *Journal of Parallel and Distributed Computing*, vol. 62, pp. 843–864, 2002.

[8] R. Hadid and V. Villain, "A New efficient tool for the design of self-stabilizing ℓ-exclusion algorithms : the controller," in *Proc. the 5th IEEE International Workshop*, WSS, 2001, pp. 137–151.

[9] M. Gradinariu and S. Tixeuil, "Tight space self-stabilizing uniform ℓ-mutual exclusion. Distributed Computing Systems," in *Proc. 21st International Conference*, 2001, pp. 83–90.

[10] T. Bernard, A. Bui, O. Flauzac, and F. Nolot, "A multiple random walks based self-stabilizing k-exclusion algorithm in ad-hoc net-works," *International Journal of Parallel, Emergent and Distributed Systems*, T. Francis eds, vol. 25, no. 2, pp. 135152, 2010.

[11] A. Cournier, A. KDatta, F. Petit, and V. Villain, "Self-stabilizing PIF algorithm in arbitrary rooted networks," in *Proc. 21st International Conference on Distributed Computing Systems (ICDCS-21)*, IEEE Computer Society Press, pp. 91-98, 2002.

[12] M. G. Gouda and F. F. Haddix, "The stabilizing token ring in three bits," *Journal of Parallel and Distributed Computing*, vol. 35, pp. 43–48, 1996.

Mehmet Hakan Karaata was born in Turkey in 1966. He received his PhD Degree in Computer Science in 1995 from the University of Iowa. He joined Bilkent University, Ankara, Turkey as an Assistant Professor in 1995. He is currently working as a Professor in the Department of Computer Engineering, Kuwait University. His research interests include mobile computing, distributed systems, fault tolerant computing and self-stabilization.

Rachid Hadid was born in Algeria in 1971. He received his PhD Degree in Computer Science in 2002 from the University of Picardie Jules Vernes, France. He worked in the University the Picardie Jules Vernes, Engineering School of Bourges, Mazoon College University, and Saad Group University as Assistant Professor from 2002 to 2010. He is currently working as Research Associate in the Department of Computer, University of Picardie Jules Vernes. His research interests include distributed systems, fault tolerant computing and self-stabilization.