A stabilizing *l*-exclusion algorithm in arbitrary rooted networks

RACHID HADID[*] AND MEHMET HAKAN KARAATA[**]

[*]*Department of Computer Engineering and Architecture, Aydin University, Besyol Mah.İnonu Cad. No:38 Sefakoy -Kucukcekmece/Istanbul, Turkish.*

[**]*Department of Computer Science and Department of Computer Eng P.O. Box 5969, Safat 13060 Kuwait*

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 ($l \geq 1$) processes, where l processes instead of 1, are free to use a shared resource simultaneously. The algorithm is semi-uniform, and its space requirement is $(l+3)\delta_r$ states (or$\lceil \log((l+3)\Delta_r \rceil)$ bits) for the root r and $4 \times \delta_p^2 \times$ D states (or $\lceil 4\log(\delta_p \times D \rceil)$ bits) for each non root process p, where δ_p is the degree of process p and D 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 the size of the network for any process, and 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 $(\lceil n/l \rceil \times (l+D))$ rounds, where n is the size of network.

Keywords: Distributed systems; fault-tolerance; self-stabilization; *l*-exclusion; propagation of information with feedback.

INTRODUCTION

In 1974, Dijkstra introduced the property of self-stabilization in distributed systems and applied it to algorithms for mutual exclusion (Dijkstra, 1974). 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 well know mutual exclusion problem where l processes are allowed to execute the critical section concurrently. A generalization of the *l -exclusion problem* is known as the k-out-of- l exclusion problem (Datta *et al.*, 2003b,a, 2011) there are l units of the shared resources, any process can request k ($1 \leq k \leq l$) of units of the shared resources, and no resource unit is allocated to more than one process at one time. The l -exclusion problem has several applications. The applications include resource sharing systems with l identical shared physical resources to be shared. In such a system, each process in a critical section may consume one unit power, and only l units of power are available. Applications of the problem also include congestion control and bandwidth allocation; again, a process in the critical section has access to a fixed quota of bandwidth, and the total bandwidth should not exceed a limit given by l. The problem was first defined and solved by Fischer, Lynch, Burns and Borodin in a generalized test and set model (Fisher *et al.*, 1979). The problem of l -exclusion has been extensively studied, number of l -exclusion algorithms are available in the literature (Afek *et al.*, 1990; Dolev *et al.*, 1988; Fisher *et al.*, 1979, 1989; Masum *et al.*, 2010; Peterson, 1990). However, to the best of our knowledge, the self-stabilizing l -exclusion algorithms are limited to (Abraham *et al.*, 2001; Antonoiu and Srimani, 2000; Bernard *et al.*, 2010; Flatebo *et al.*, 1994; Gradinariu and Tixeuil, 2001; Hadid, 2002; Hadid and Villain, 2001; Villain, 1999). Dijkstras self-stabilizing solution of the mutual exclusion problem (Dijkstra, 1974) was generalized in (Flatebo *et al.*, 1994). In that paper, the authors present a self-stabilizing l -exclusion algorithm working in the state model and uses the token-ring communication topology. The second self-stabilizing l -exclusion algorithm was published in (Abraham *et al.*, 1997, 2001), but unlike the first solution the algorithm utilizes a set of single-writer multiple-reader regular registers as shared memory. In both cases, the space requirement depends on the size of the network and l. Algorithms in (Flatebo *et al.*, 1994; Abraham *et al.*, 1997, 2001) require $\Omega(n^2 l)$ and $O(2^n)$ states per process, respectively, where n is the size of the network. Algorithm in (Abraham *et al.*, 1997, 2001) also assumes a knowledge of l. Algorithm (Antonoiu and Srimani, 2000) works on trees, uses the state model, and requires $O(Max^{2\delta+1})$ states per process, where δ is the degree of the network and $Max \geq l$. First attempts to solve the l -exclusion problem in state model with a space complexity independent of n (and almost independent of l) are presented in (Villain, 1999) for rings and (Hadid, 2002) for trees. All those

algorithms run in the state model (Flatebo *et al.*, 1994; Hadid, 2002; Villain, 1999) or in the shared memory model (Abraham *et al.*, 1997, 2001). The algorithm in (Hadid and Villain, 2001) presents the first self-stabilizing *l*-exclusion in message passing model in ring and tree networks. All those algorithms are semi-uniform and deterministic. In (Gradinariu and Tixeuil, 2001), two randomized uniform solutions in unidirectional rings are presented using state model. The first algorithm requires $(l \times log(n))$ states per process and the second algorithm requires $(l \times \log^2(n))$ states per process. Recently, (Bernard *et al.*, 2010) proposed a random walk solution in message passing model in ad hoc networks. The drawback of this kind of solution is that the waiting time of process to enter the critical section is not bounded.

Contribution. In this paper, we present the first self-stabilizing *l*-exclusion algorithm in arbitrary networks. The proposed algorithm uses a variant of the well-known Propagation of Information with Feedback (PIF) scheme, called Propagation of information with Feedback and Cleaning (PFC) introduced in (Cournier *et al.*, 2002). The proposed algorithm is token-based, i.e., a process can enter its critical section only upon receipt of a token. In the proposed algorithm, the token distribution is initiated by the root, where each process distributes available tokens in the breadth first manner, i.e., tokens are passed to different neighbors (provided that more than one neighbor exists) following a local ordering. The proposed algorithm is an extension of the approach introduced in (Hadid, 2000) to arbitrary networks. Although the proposed solution has some similarities with our previous work, such as the use of tokens and PFC scheme, the current work has many significant contributions. In the current work, we do not assume a pre-constructed spanning tree using existing solutions in the literature (Gartner, 2003). Instead, we use explore the token distribution phase and dynamically constructing the spanning tree along with this distribution. Thus, we do not need to wait the spanning tree construction to start the token distribution. Also observe that, since we consider asynchronous scheduler instead of a synchronous scheduler, while the tree is dynamically constructed by the token distribution, some segment of the network may make more progress in the token distribution than others. In addition, unlike in synchronous systems where progress is made at synchronous steps, in our asynchronous algorithm, progress is made whenever a guard is enabled without having to wait for the next synchronous step. As a result, the token distribution may not follow the shortest path in the network and the execution of our algorithm on different networks, will lead to different trees. The space requirement of our algorithm is $(l+3)\delta_r$ states (or$\lceil \log((l+3)\delta_r \rceil)$ bits) for the root r, and $4 \times \delta_p^2 \times$ D states (or $\lceil 4 \log(\delta_p \times D \rceil)$ bits) for any non root process p, where δ_p is the degree of process p and D is the diameter of the network. The diameter of a network is the largest distance between any two nodes in the

network, where the distance between any two nodes is the minimum number of hops between the nodes. This is the first l-exclusion algorithm on arbitrary networks in which the space requirement is independent of the size of the network for any process, and is independent of l for all processes except the root. Our algorithm is self-stabilizing and its stabilization time is O $(\lceil n/l \rceil \times (D + l))$ rounds. In addition, the proposed algorithm adapts to topology changes in the form of process/link failures and additions as long as the topology remains connect and root process does not crash. So, upon a topology.

Outline of the paper. The rest of the paper is organized as follows. In Section 2, 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 3, and its correctness proof in Section 4. Finally, we make some concluding remarks in Section 5.

PRELIMINARIES

Distributed System. We consider a distributed system as an arbitrary undirected connected graph G = (V; E), where V is the set of nodes ($|V| = n$) and E is the set of edges. Nodes of G represent *processes*, and edges of G represent *bidirectional communication links*. Two nodes connected by a communications link are said to be neighbors. We consider networks which are *anonymous*; i.e., no process, except the root (identified by r), has any identity. However, each process uses an id to index each of its neighbors, where the id indicates the order of the neighbor among the neighbors of the process. We consider networks which are *asynchronous*: there are bounds neither on communication delays, nor on process speeds. A communication link (p,q) exists if and only if p and q are neighbors. For convenience, we assume that each process p labels its links 1, 2,...,δp, where 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.

Programs. 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. A process can read its own variables and those owned by the neighboring processes. Each action is of the following form: $< label >:: < guard > \rightarrow < statement >$. 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**, the corresponding action is said to *enabled*; disabled, otherwise. A process is called *enabled* if it has at least one action enabled. If multiple actions at

process are enabled at any time, an enabled action is selected non deterministically and executed atomically. 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 be the set of all possible configurations 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 as E. We assume a *weakly fair and distributed daemon* to simplify the presentation of the implementation of the algorithm and its correctness. The *weak fairness* means that if a process p is continuously enabled, then p will be eventually chosen by the daemon to execute an action. The *distributed* daemon implies that during a computation step, if one or more processes are enabled, the daemon chooses at least one (possibly more) of these enabled processes to execute an action. In order to compute the time complexity measure, we use the definition of *rounds* (Dolev *et al.*, 1997). This definition captures the execution rate of the slowest process in any computation. Given a computation e ($e \in E$), the *first round* of e (let us call it e′) is the minimal prefix of e containing one (local) atomic step of every continuously enabled process from the first configuration. Let e″ be the suffix of e, i.e., $e = e'\, e''$. Then the *second round* of e is the first round of e'', and so on.

Privilege. The definition of *privilege* to enter the critical section is the same as in (Gouda and Haddix, 1996; Hadid, 2002): A process 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 process is privileged if it holds a token.

Specification of the *l*-exclusion protocol. We consider a computation e of P to satisfy SP_P 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 $k < l$ processes execute the critical section forever and some other processes are requesting the critical section, then eventually at least another process will enter the critical section.

Self-stabilizing l-exclusion Protocol. 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. This specification

is referred to as high-level *l*-exclusion and the specification involving only safety and fairness is referred as low-level *l*-exclusion. Analogous to all existing self-stabilizing solutions to this problem in the literature, except (Gradinariu and Tixeuil, 2001; Hadid and Villain, 2001), the proposed algorithm cannot ensure that every computation of our algorithm always satisfies the liveness property (i.e., low-level *l*-exclusion algorithm): some processors may have to wait for others which are in their critical section, even if the total number of processors in the critical section is less than *l*.

SELF-STABILIZING L -EXCLUSION ALGORITHM.

In this section, we present the stabilizing *l*-exclusion algorithm (Algorithm 3.1) with $\delta_r > 1$ (the case of $\delta_r = 1$ is a simple version of this algorithm).

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 start the PIF phase to clean up the network from the distributed tokens. The clean up phase (PIF) follows but does not meet the token distribution phase, 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 phase every process records the process from which it receives a token as its parent, a tree rooted at the root is gradually built. 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 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 distrib.

Algorithm 3.1 Self-Stabilizing ℓ-Exclusion Algorithm

Input N_p: set of (locally) ordered neighbors;
$D = n - 1$;

Constants For the root $L_p = 0$; $P_p = p$;

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, $P_p : \{1, ..., \delta_p\}$; $L_p : 0, ..., D$;

Macro

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

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

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

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

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

Actions

For the root node

Predicates

$1\text{st}Token(p) \equiv (T_p = C) \wedge (\forall q \in N_p :: T_q = C)$

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

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

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

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

$(a_1) :: 1\text{st}Token(p) \rightarrow S_p := Next_p; T_p := 0;$ **(PR)** /* Token distribution and Cleaning */

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

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

$(a_4) :: Broadcast(p) \rightarrow T_p := B;$ /* PIF Synchronization */

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

For other nodes

Predicates

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

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

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

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

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

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

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

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

$(a_7) :: Broadcast(p) \rightarrow T_p := B;$ /*PIF Synchronization and Token Clean*/

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

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

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

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

When the root receives feedback from all its children, it knows that the tree is cleaned up from the previous tokens and starts a new token distribution phase. The repetition of 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 3.1.

The token distribution phase is implemented using the privileged actions a_1, a_2, and a_6, which are marked as "(PR)". So p is privileged or has a token iff one of the following conditions holds: p is the root ($p = r$) and a_1 or a_2 is enabled, or p is non-root process and a_6 is enabled. The cleanup process (PIF) is implemented using actions a_4 and a_5 for the root, and 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. $T_p \in \{0, ..., l\text{-}1, B,C,R\}$ for the root process and $T_p \in \{Tok,B,C,R\}$ 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 $(q \in Np)$ 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.

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 process 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 such a neighbor 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 it is 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, if the root is also a potential parent of p, then p chooses to receive the token from the root process; otherwise, p chooses the neighboring process with the smallest link number (Macros Par_p and $Potential_p$). 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, ..., l-1* of the variable T_r to differentiate the distribution of its l tokens. A non-root process q receives a token when the following conditions hold: the T_q variable of process q is in state C; one of the neighbors p of process q, such that $T_p = Tok$ *(or* $\in \{0,\ldots, l\text{-}1\}$ if p is root) holds, has selected q as its potential child by assigning q to its S_p variable; and the T variable of next process to receive the token from process q, if such a process exists, is equal to C. When a leaf process assigns *Tok* 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 process 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 chosen a parent yet, then q is involved in this phase as follows (a_6):

(i) If q has no parent, then it chooses its parent p by assigning the link number associated with p to its variable P_q (consequently, q becomes a child of p). This leads to process q joining the tree rooted at r.

(ii) decides its level by assigning $L_p + 1$ to its variable L_q,

(iii) selects the next process (if any) by advancing S_q 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 transmit its token to, then it destroys the token and becomes leaf process in the tree, and

(iv) q passes its token by changing its T value to Tok. When p uses the token by

executing a_6, p cleans the trace of this token (Tok 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 $(l\text{-}1)^{th}$ token) (a_3).

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 cleanup process is implemented using the PIF scheme. To implement this phase we specifically use the PFC introduced in (Cournier *et al.*, 2002). This scheme requires some additional values and variables. $T_p = B$ and $T_p = F$ refers to the *broadcast* and *feedback* state, respectively. The root uses another additional value 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 the 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, the root starts a new PFC by switching its T_r variable to B (a_4). When process $p \in V\{\text{-}r\}$ with $T_p = C$ discovers that the T variable of its parent process has been set to B, then p participates in the broadcast phase and changes its T_p variable to B (a_7). When the broadcast phase reaches a leaf process, the leaf process knows that all its ancestors have entered the broadcast phase. The leaf process then starts the feedback phase by assigning F to its T variable (a_8). The feedback phase propagates towards the root in a bottom-up manner as follows. Each internal process p which finds all its children in state F, participates in the feedback phase by assigning F to its T variable (also a_8). Eventually, the feedback phase reaches 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 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 the value of its T_p variable only if it has the value of its Tp variable set to C, all the children of process p have their T_p variable set to C, and the parent P_p of process p has its T_p variable set to B (see action a_7).

Illustrative example

We now describe the behavior of the algorithm using Figure 3.1 and we consider a situation of 3-exclusion. For simplicity, we assume that the tree is not built, i.e., P variables are sited to $\perp$, and all processes are in the cleaning phase, i.e., T values are sited to C.

The initial configuration is shown in Part (*i*) where the root is ready to launch the token distribution phase. In Part (*ii*), r chooses the neighbor a to receive its first token ($Macro\ Next_r$), sets its T_r variable to 0, and points its S_p value to the link number associated to process a. During its turn, a sends the received token to its neighbor e (thus causing process e to take process a as its parent), sets its T_a value to Tok, and points its S_p value to the link number associated to e (Part (*iii*)). Next, the root, seeing that a received the token it sent, sends the second token to b, while e sends the token to its neighbor i and sets T_c to Tok (Part (*iv*)). Now both i and b hold a token. Process b sends its token to f and sets T_b to Tok. However, process i destroys its token since it is a leaf process (Part (*v*)). During this step, a changes T_a to C (i.e., cleans the trace of the token) to be ready to receive another token or participate in the PIF phase. In Part (*vi*), the root, seeing that b received the token it sent, sends its last token to c, while f destroys the token it receives from b and sets T_c to Tok. During this step, e changes T_e to C (i.e., cleans the trace of the token). Next, c sends its token to g and sets its T variable to Tok (Part (*vii*)). During this step, both i and b change their T variable to C and clean the trace of the token).

Note that in a 4-exclusion algorithm, the root would have a fourth token to send to its next neighbor before initiating the PIF wave and since $T_d = C$, the root could send this last token to d.

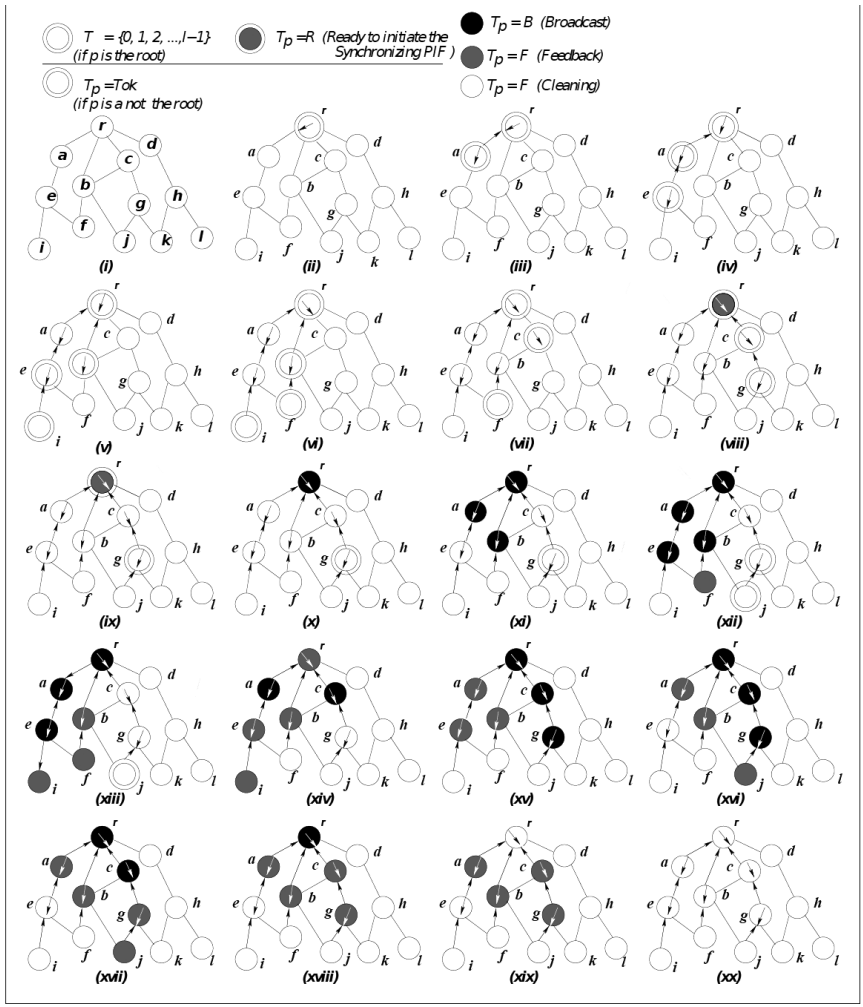

Fig. 3.1. An example showing the normal execution of the Algorithm

However, since we dealing with 3-exclusion in this example, and the root has sent its third token, the root now initiates a PIF wave to clean up the tree from the tokens and decide when it can start a new distribution of tokens. Hence, r changes its T value to R (Part (*viii*)). This enables all its children to change their T values to C. During this step, f changes T_f to C (i.e., cleans the trace of the token) and process g sends its token to j and sets T_g to Tok. Then, c changes its T_c to C (Part (ix)). In Part (*x*), all the children of the root have changed to C. Then, r initiates the PIF wave. The PIF wave should not be allowed to meet any token, i.e, processes with T equal to B must not be allowed to confuse the processes with *T equal to Tok*. In other words, the PIF wave cannot disturb the forwarding of the tokens. We implement this constraint as follows: A process p

can change T_p to B only if T_p has a value C, all of its children have the C value, and P_p has the value B. This is shown in Part (x). processes a and b change the value of their T_p *variable to* B, whereas process c changes the value of it T_p variable to B only after g changes its T_p variable to C. Eventually, the broadcast of the PIF wave reaches the leaf processes. In the next step, the leaf processes can initiate the feedback phase. This is shown in Part (*xii*). The leaf process f initiates the feedback phase (changes to F), and in Part (*xiii*), i also initiates the feedback phase. Now, any internal process can change its T value to F (so, participates in the feedback phase) as soon as all its children have changed their T values to F (i.e., have completed the feedback phase) and its parent has changed its T variable to B (Part (*xii*) and (*xvi*)). Next, every process can clean its T value (Part (*xiv*) and (*xv*)). So, the feedback and cleaning phase can run concurrently (Part (*xv*)). All we have to make sure is that the cleaning phase does not meet the broadcast phase, i.e., the processes in the cleaning phase do not confuse the processes in the broadcast phase. We implement this constraint as follows: An internal process p can change its T value from F to C only if all its neighbors in the tree are in the feedback or cleaning phase (i.e., has their T values $\in$ {F; C}). When the feedback phase reaches all the children of the root, the root terminates the feedback phase and changes its T value to C. Now, all its children can change their T value to C. So, the root will remain locked until all its children change to C. This configuration is equivalent to Part (*i*). This marks the end of the current cycle and the start of the next cycle. As the system is asynchronous, the processes may not change their states at the same time. We need to make sure that the forwarding of the tokens of the next cycle does not confuse the processes in the feedback phase. The root (resp., an internal process) can initiate the next cycle (resp., can participate to the token distribution) only if all its children change from F to C (Part (*xv*)).

Error Correction.

During 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 (see Figure 3.2).

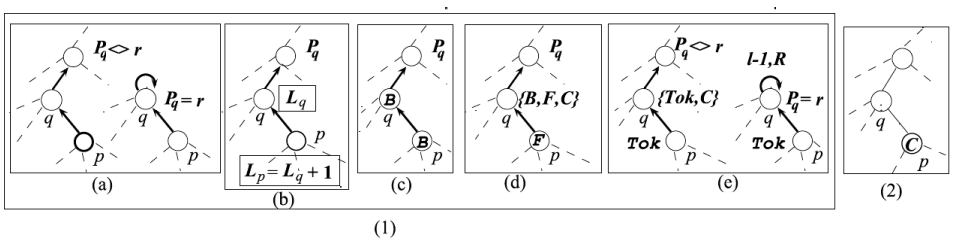

Fig. 3.2. An example showing the process's normal states

(1) For each process p which has already chosen its parent (i.e., $P_p = q$), the following properties need to be maintained.

(a) The parent q of p has also chosen its parent, i.e., $P_q \neq \perp$.

(b) The distance L_p of process p is one plus that of the parent, i.e., $L_p = Lq + 1$.

(c) If a process p is in the broadcast phase, then its parent q is also in the broadcast phase.

(d) If a process p is in the feedback phase, then its parent q is either in the broadcast, cleaning, or feedback phase.

(e) If a process p is in a token distribution phase (i.e., $T_p = Tok$), 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) Each process p which has not yet chosen its parent (i.e., $P_p = \perp$) is in a cleaning phase.

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* (Predicates *AbnPath(p)* and *AbnPif(p)*). For satisfying these properties, the correction actions a_{10} and a_{11} (Algorithm 3.1) are used. The correction action a_{10} is used when property 1 holds for process p while one (or both) of the properties (a) and (b) does not hold. However, action a11 is used when either property 1 holds for p and one (or more) of the properties (c), (d) or (e) does not hold, or property 1 does not hold for p and p is not in a cleaning phase (i.e., property 2 does not hold for p).

PROOF OF CORRECTNESS

We are to show that starting in any arbitrary system configuration, the proposed algorithm implements a stabilizing l-exclusion algorithm (Theorem 4.5). We first show that the proposed protocol is guaranteed to eventually enter a *normal system configuration* in which no process exists in an abnormal state (Theorem 4.1). Then, we show that the root starts the token distribution and the PIF phases infinitely often (Lemma 4.10). The repetition of the token distribution phase in a fair manner ensures that every process will get a token (fairness) (Theorem 4.3) and the complete spanning tree of the network is eventually completed (Lemma 4.11). The safety is clearly obtained from the fact that the token distribution starts only after cleaning the tree from the tokens and the root distributes exactly l tokens in the tree (Theorem 4.2).

We need the following definitions to facilitate the remainder of the proof of

correctness. A path μ_{pq} is a sequence $p = p_1, p_2, ..., p_k = q$ such that *(i)* $k \geq 2$, *(ii)* $\forall_i, 1 < i \leq k$, $P_{pi-1} = p_i$, *(iii)* $p_1 = r$ *or* $P_{p1} = \perp$. Process p_i, $\forall_i$, $1 < i \leq k$, $P_{pi-1} = P_i$, is said to *belong* to a path μ_{pq} and is denoted as $p_i \in \mu_{pq}$. The *length* of the path μ_{pq} is denoted by $\overline{\mu_{pq}}$. If $p_1 = r$ the μ_{pq} is called a routed path; a non-routed path otherwise. Let $(T_{p=p_1}, T_{p_2}, ..., T_{p_k=q})$ denote a state sequence of a maximal path of processes $p = p_1, p_2, ..., p_k = q$, such that process p_1 is the parent of process p_2, process p_2 is the parent of p_3 and so on, in configuration γ. A subsequence T_pT_q of length two of a state sequence is referred to as *abnormal state pair* if process q is in an abnormal state and process p is the parent of process q; a *normal state pair*, if q is in normal state. Based on the definition of normal and abnormal process, observe that a state pair T_pT_q is an abnormal state pair iff $P_p = \perp$, or $L_q \neq L_p + 1$, or $T_pT_q \in \{BTok, FTok, FB, xB, xF, CB\}$, where $x \in \{Tok, R, 0, 1, ..., l\text{-}1\}$. A state pair T_pT_q is a normal state pair iff $P_p \neq \perp$, $L_q \neq L_p + 1$, and $T_pT_q \in \{BC, BB, BF, FC, FF, CF, yTok, yC\}$, where $y = x \cup \{C\}$. A state sequence $(T_{p=p0}, T_{p=p1}, ..., T_{pk=q})$ is referred to as *abnormal sequence state* if it contains an abnormal state pair. For any process p, we define a set *Tree(p)* of processes as follows: For any process q, $q \in$ *Tree(p)* i p is the terminus of $\mu(pq)$. We denote the height of *Tree(p)* by *h(Tree(p))*.

Removal of abnormal state

In this section we establish that each abnormal state sequence eventually disappears. In our algorithm, actions a_{10} and a_{11} are used to remove the abnormal state pairs.

Lemma 4.1. *For every process p, if* $P_p = q$ *holds, then* $L_p = L_q + 1$ *or* $P_p = \perp$ *holds in at most one round.*

Proof. Let p be a process such that $P_p = q$ and $L_p \neq L_q + 1$. Then in at most one round p executes a_{10} sets its $P_p = \perp$.

Lemma 4.2. *Starting from any configuration, every non-routed path disappears in at most D rounds.*

Proof. Let $\mu_p q = p_1, p_2, ..., p_k$ be a non-routed path, i.e., $p_1 \neq r$. We show this result by induction on $\overline{\mu_{pq}}$.

Basic Step. Let $\overline{\mu_{pq}} = 1$. So, $\mu_{pq} = p_1, p_2$. Then, in at most one round p_2 executes a_{10} and sets its P_{p2} to $\perp$ as the result follows.

Induction Step: Assume that the result is true for $\overline{\mu_{pq}} \leq k - 1$, for $k > 1$. Consider the case where $\overline{\mu_{pq}} = k$. By induction hypothesis, $\overline{\mu_{pq}} = p_1, p_2, ..., p_{k-1}$ disappears in at most k - *2* rounds, i.e., $\forall_i$, $1 < i \leq k\text{-}1$, $P_{pi} = \perp$ holds after at most k - *2* rounds. Then, in at most one round process p_k executes a_{10} and sets its P_{pi} to $\perp$. Since the length of the path μ_{pq} is bounded by D the result follows.

Lemma 4.3. For every process p, *if* $P_p = \perp$ *then* $T_p = C$ *holds in at most one round.*

Proof. Let p be a process such that $P_p = \perp$ and $T_p \neq C$. Then in at most one round p executes a_{11} and sets its $T_p = C$.

Lemma 4.4 *Starting in any arbitrary system configuration, no state sequence contains an abnormal state pair in the set* $\{BTok, FTok, FB, xB, xF\}$ *(where* $x \in \{T_{ok}, R, 0, 1, \ldots, l\text{-}1\}$*) in at most one round.*

Proof. Let $T_{Pp}T_p$ be a state pair such that $T_{Pp}T_p \in \{BTok, FTok, FB, xB, xF\}$ *(*$x \in \{Tok, R, 0, 1, \ldots, l\text{-}1\}$*). As* $\neg$ *Normal(p)* holds for process p, process p executes a_{11} and changes its T value to C in at most one round. Then, immediately after process p changes its T value to C, $T_{Pp}T_p \in \{BC, FC, FC, xC\}$ holds. Observe that when process P_p or process p in state pair $T_{Pp}T_p \notin \{BT_{ok}, FTok, FB, xB, xF\}$ executes an action, no state pair in set $\{BTok, FTok, FB, xB, xF\}$ *(*$x \in \{Tok, R, 0, 1,, l\text{-}1\}$*) is created.*

Lemma 4.5 *Starting in any arbitrary system configuration, no state sequence contains the abnormal state pair CB in at most D rounds.*

Proof. Let $T_{Pp}T_p$ be a state pair such that $T_{Pp}T_p = CB$. Notice that since $\neg Normal(p)$ holds for process p, process p executes a_{11} and changes its T value to C in at most one round. If $Child_p = \phi$, then the abnormal state pair disappears and the result follows. Otherwise, let q be a process such that $q \in Child_p$. From Lemma 4.4, $T_pT_q \in \{BC, BB, BF\}$. Let $\gamma_{i-1} \mapsto \gamma_i$ be a computation step in a round, such that in this step, process p executes a_{11}. It is clear that the abnormal state pair $T_{Pp}T_p$ becomes normal immediately after process p executes a_{11} and changes its T value to C. Then, two cases are possible: either the abnormal state pair disappears from the system or another abnormal state pair (i.e. T_pT_q) is formed after the execution of action a11.

1. We consider the case where the state pair T_pT_q remains normal state pair in γ_i. It is easy to see that immediately after the computation step $\gamma_{i-1}, \mapsto \gamma_i$, process q becomes normal. Hence, $T_{Pp}T_p = CC$ and $T_pT_q \in \{CC, CF\}$. Thus, the abnormal state disappears in one round.

2. We consider the case where the state pair T_pT_q becomes abnormal state pair in γ_i when process p assigns C to its T value by executing action a_{11}. Notice that immediately after the computation step $\gamma_{i-1} \mapsto \gamma_i$, process p becomes normal, whereas process q becomes abnormal. Hence, $T_{Pp}T_p = CC$ and $T_{Pp}T_p = CB$. Therefore, the abnormal state pair $T_{Pp}T_p$ becomes normal state pair and the normal state pair T_pT_q becomes abnormal state pair. Consequently, the path distance of the abnormal state pair sequence increases by one after this computation step $\gamma_{i-1}, \mapsto \gamma_i$.

Since the path distance of abnormal state sequence is bounded by D, the result follows by repeating Cases 1 and 2.

By Algorithm 3.1, no extra abnormal process is created. Thus, we can claim the following result.

Corollary 4.1. *If there exists no abnormal process in γ_i, then there also exists no abnormal process in γ_i, for any $j > i$.*

The following theorem shows that eventually a *normal system configuration* is entered. A system configuration? is referred to as a normal configuration if and only if it contains no process in an abnormal state. The theorem follows directly from Corollary 4.1 and Lemmas 4.1, 4.2, 4.3, 4.4, and 4.5.

Theorem 4.1. *Starting from any initial configuration, the system enters a normal configuration in at most $D + 1$ rounds, and the system configuration remains normal thereafter.*

Convergence and Correctness

In this section, we will focus on the definition of the *number of tokens* in the whole network. From Theorem 4.1, we assume that there is no abnormal process in the network (normal configuration). We denote by *Token(p)* if a privileged rule of p is enabled or not. The notion of token as an enabled privileged rule is not sufficient to enumerate the number of tokens of the system. For example, the tokens left at the root will eventually circulate. So, they have to be included in the computation of the total number of tokens in the network. Another situation is the following: $(T_p = C) \wedge (T_{Pp} \in \{Tok, 0, \ldots, l\text{-}1\}) \wedge (S_{Pp} = p) \wedge (T_{Nextp} = Tok)$, where p does not have the token since it cannot apply a_6, but it will eventually have a token as soon as $Next_p$ assigns C to its T variable. In this case, we say that p has a potential token and we denote it as PToken(p). We denote by T_{potr} the potential number of tokens which accumulates at the root. So,

$$T_{pot_r} = \begin{cases} \ell & if\ T_r = C \wedge Token(r) \\ \ell - 1 & if\ T_r = GToken(r) \\ \ell - T_r - 1 & if\ T_r \in \{0, \ldots, \ell - 1\} \wedge\ Token(r) \\ \ell 0 - T_r - 2 & if\ T_r \in \{0, \ldots, \ell - 1\} \wedge Token(r) \\ 0 & otherwise \end{cases}$$

We denote T_{potI} the potential number of tokens which could be at the non-root processes. So, $T_{potI} = \#\{p: PToken(p)\}$ such that PToken(p) $(T_p = C) \wedge TPp \in \{T_{ok}, 0, \ldots, l\text{-}1\}) \wedge (S_{Pp} = p) \wedge (T_{Nextp} = Tok)$.Thus, the number of tokens in the whole network is denoted by T_{net} and is equal to $(T_{potr} + T_{potI} + \#\{p: Token(p)\})$. Observe that in any initial normal configuration, the number of tokens in whole network is arbitrary. We show that the system enters a

normal configuration when $T_{net} \leq l$ holds and continues to hold thereafter.

In Lemmas 4.8 and 4.10, we show that starting from any configuration; the root sends all its tokens and initiates a PIF infinitely often. Then, in Lemma 4.11, we show that every process will join the tree *Tree(r)*. In Lemma 4.12, we show that after initiating each PIF, we reach a configuration where the number of tokens in *Tree(r)* is null ($T_{net} = 0$). Then, starting from this configuration, the root sends always l tokens and the number of tokens in *Tree(r)* is less or equal to *l* forever. We first need to show Lemmas 4.6 and 4.7.

Lemma 4.6 *If PToken(p) holds for process p, then Token(p) holds in at most h(Tree(p)) + 1 rounds.*

Proof. Let p be a process such that *PToken(p)* holds for process p. So, the condition $(T_p = C) \wedge T_{Pp} \in \{T_{ok}, 0, \ldots, l\text{-}1\}) \wedge (S_{Pp} = p) \wedge (T_{Nextp} = Tok)$ holds. Let $q = Next_p$. Obviously, $P_q = p$ since $T_q = Tok$. We will show that $T_q = C$ (hence, *Token(p)*) holds in at most $h(Tree(p)) + 1$ rounds. We show this result by induction on $h(Tree(p))$.

Basic Step: Let $h(Tree(p)) = 1$. So, q is a leaf process. We consider two cases.

1 - $S_q = \perp$. Then, in at most one round, q executes a_9 and changes its T value to C. Hence, the result follows.

2 - $S_q \neq \perp$. So, there exists $q' \in N_q$ such that $T_{q'} = C$ and $P_{q'} = \perp$. Then, in at most one round, q' changes its T value to *Tok* and points its $P_{q'}$ to q. Subsequently, in at most one round, q executes a_9 and changes its T value to C and the result follows.

Induction Step: Assume that the result is true for $h(Tree(p)) \leq k\ 1,\ k > 1$. Consider the case where $h(Tree(p)) = k$. We need to consider two cases.

1 - $T_{Sq} = Tok$. Then, in at most one round, process q executes a_9 and assigns C to T_q. Hence, the result follows.

2 - $T_{Sq} = C$. We need to consider two cases.

a) *Token(S_q)*. Then, in at most one round, S_q executes a_6 *and changes its T* value to Tok. Subsequently, in at most one round, q executes a_9 and changes its T value to C and the result follows.

b) *PToken(S_q)*. By induction hypothesis, *Token(S_q)* holds within at most $h(Tree(S_q)) + 1$ rounds. Then, we reach Subcase (a). So, the result follows in at most two rounds

Lemma 4.7 *For every process, if* $(T_p = T_{ok}) \wedge (S_{Pp} \neq p \vee T_{Pp} \notin \{Tok, 0, \ldots, l\text{-}1\})$ *holds for p, then* $T_p = C$ *holds in at most* $h(Tree(p)) + 2$ *rounds.*

Proof. Let p be a process such that $(T_p = T_{ok}) \wedge (S_{Pp} \neq p \vee T_{Pp} \notin \{Tok, 0, ..., l-1\})$, we need to consider two cases.

a) $Token(S_p)$. Then, in at most one round S_p executes a6 and changes its T value to Tok. Subsequently, in at most one round, p executes a_9 and changes its T value to C. Hence, the result follows.

b) $PToken(S_p)$. From Lemma 4.6, $Token(S_p)$ holds within at most $h(Tree(S_p)) + 1$ rounds. Then, we reach Case 1.

Lemma 4.8 *The root process will send all its potential tokens in at most* $h(Tree(r)) + 2 \times l - 2$ *rounds.*

Proof. Consider the following two cases.

1 - $T_r = C$. We will show that $\forall p \in N_r, T_p = C$ holds in at most $h(Tree(r)) + 1$ rounds. Assume that there exists $p \in N_r$ such that $T_p \neq C$. So, $T_p \in \{F; Tok\}$.

(a) $T_p = F$. Then, in at most one round, p executes a_9 and changes its T value to C.

(b) $T_p = Tok$. From Lemma 4.7, $T_p = C$ holds in at most $h(Tree(p)) + 2$ rounds.

2 - $T_r \in \{0, ..., l-2\}$. Let $p = S_r$ and $q = Next_r$. We will show that $T_p = Tok$ (Case a) and $T_q = C$ (Case b and c) holds in at most $h(Tree(r)) + 1$ rounds so that the root is enabled to send its next token. We have $p \in \{C, Tok\}$ and $q \in \{C, Tok, F\}$.

(a) If $T_p = C$, then from Lemma 4.6, $Token(p)$ holds in at most $h(Tree(p)) + 1$ rounds. Then, after one round, p executes a6 and changes its T value to Tok.

(b) If $T_q = F$, then in at most one round, q executes a_9 and changes its T value to C.

(c) If $T_q = Tok$, then from Lemma 4.7, q will change its T value to C within at most $h(Tree(q)) + 2$ rounds.

Thus, $T_p = Tok \wedge T_q = C$ holds in at most $h(Tree(r)) + 1$ rounds. Then, in at most one round r executes a_2, increases it T value by one, and points its S_p to q. Subsequently, q changes its T value to Tok and Nextr changes its T value to C in at most one round. By repeating this argument, we reach a configuration where $T_r = l - 1$ in at most $2 \times (l - 2) + 1$. Hence, the results follows after at most $2 \times l + h(Tree(r)) - 2$ rounds.

Lemma 4.9 *For every process p, if* $T_p = B$ *holds and continues to hold, then* $\forall q \in Child_p$; $T_q = F$ *holds in at most* $3 \times h(Tree(p))$ *rounds.*

Proof. Let p be a process such that $T_p = B$. In the worst case, we have a configuration where $(T_p = B) \wedge (\forall p_0 \in Child_p :: T_{p0} = C)$. In such a configuration, p is not enabled until all its children change to F (a_8). Let $p_0 \in Child_p$, if $(\forall p_0 \in Child_p :: T_{p_1} = C$, then p_0 can immediately forward the PIF (i.e., executes a_7 and changes its T value to B) to its children. Otherwise $\exists p_1 \in Child_{p0} :: T_{p_1} \neq C (T_{p1} \in \{T_{ok}, F\})$. So, p_0 is not enabled until p_1 changes its T value to C. If $T_{p1} = F$, then p_1 changes its T value to C after one round. Otherwise, i.e., $T_{p1} = Tok$, then by Lemma 4.7, p_1 will change its T value to C in at most $h(Tree(p1)) + 2$ rounds. Subsequently, p_0 changes it T value to B in at most one round and simultaneously all the children of p_1 change also their T values to C. By repeating the same argument on the processes down the tree $Tree(p)$, the leaves of $Tree(p)$ will change to B in at most $h(Tree(p1))$ rounds. So, the B value is propagated through the tree $Tree(p)$, i.e., from process p to the leaves, in at most $2 \times h(Tree(p))$ rounds. Once a leaf process changes its T value to B, it initiates the *feedback* phase, i.e., changes its T value to F (a_8), in at most one round. Then, all the internal processes participate by forwarding the feedback message to process p. The feedback message reaches all the children of process p in at most $h(Tree(p))$ rounds. Since during this phase some processes can change their T value to C, the system reaches a T-normal configuration $\gamma \mapsto (\forall p \in Tree(p) :: v_{rp} \in BF^+ = F, C^*)$.

Lemma 4.10 *The root process initiates a PIF and sends all its potential tokens (infinitely often) in at most* $5 \times h(Tree(r)) + 2 \times l + 4$ *rounds.*

Proof. We need to consider the three following cases.

1 - $T_r \in \{C, 0, 1, , l\text{-}1\}$. By Lemma 4.8, root sends all its tokens and we reach a configuration where $T_r = R$ after at most $h(Tree(r)) + 2 \times l$ rounds.

2 - $T_r = R$. We will show that $\forall p \in N_r, T_p = C$ holds after at most $h(Tree(r)) + 2$ rounds. Assume that there exists $p \in N_r$, such that $T_p \neq C$. So, $T_p \in \{F, Tok\}$ rounds.

   (a) $T_p = F$. Then, in at most one round, p executes a_9 and changes its T value to C.

   (b) $T_p = Tok$. From Lemma 4.7, $T_p = C$ holds in at most $h(Tree(p)) + 2$ rounds. Then, in at most one round, r changes its T value to B (a_4).

3 - $T_r = B$. From Lemma 4.9, we reach a configuration where $\forall p \in Child_r$, $T_p = F$ holds after at most $3 \times h(Tree(r))$ rounds. Then, r changes to C in at most one round.

Lemma 4.11 *For every process p, the condition* $p \in Tree(r)$ *holds in at most* $(\lceil n / l \rceil \times (5 \times h(Tree(r)) + 2 \times l + 4)$ *rounds.*

Proof. From Lemma 4.10, the root initiates tokens infinitely often. Obviously, by the token distribution mechanism used in our algorithm, every process p will be visited by a token (hence, $p\, p \in Tree(r))$ after the root sends a set of n tokens (n is the number of nodes in the network). It is clear that if $l \geq n$ then every process will receive a token after each wave of tokens. Otherwise, after the root sends $\lceil n/l \rceil$ waves of tokens, each process will receive at least one token. By Lemma 4.10, each wave of tokens takes at most $5 \times h(Tree(r)) + 2 \times l + 4$ rounds. So, every process will be visited by a token (i.e., causing the process to join the tree $Tree(r)$ after at most $\lceil n / l \rceil \times (5 \times h(Tree(r)) + 2 \times l + 4$ *rounds*.

Starting from a normal configuration where $\forall p :: p \in Tree(r)$, the root eventually initiates (a_4) the first PIF (Lemma 4.10), which is the propagation of B-segments from the root to the leaves on $Tree(r)$. All the internal processes participate by forwarding the PIF to the leaf processes (a_7). When the PIF wave reaches the leaves, the leaf processes initiate the *feedback* phase (a_8), which is the propagation of F -segments from the leaves to the root. The internal nodes participate by forwarding the *feedback* message to the root (a_8). During this phase, some processes can erase (clean) the trace of this phase by changing their T value to C (a_9). The *feedback* phase ends when it reaches all the children of the root. Thus, the system reaches a normal configuration: $\gamma : \gamma \mapsto (\forall p \in Tree(r) :: \mu_{rp} \in BF^{+}\ \{F, C\}^*)$. It is clear that, from a configuration, the root is enabled to terminate the current PIF (a_5). Then, the root has l potential tokens, and all its children p are enabled to change to C (a_9). As explained above, the root can initiate the next token distribution (a_1) only if all its children changed their T value from F to C. So, the initial normal configuration INC can be defined as follows:

$$\gamma \in INC \Longleftrightarrow \gamma \mapsto (\forall p \in Tree(r) :: \mu_{rp} \in \{C\}\{C\}\{F, C\}^*).$$

Lemma 4.12 *Starting from any configuration, the system will eventually reach a configuration* $\gamma \in INC$ *in at most* $O(\lceil n / l \rceil \times (l + D))$ *rounds.*

Proof. From Theorem 4.1, the system enters a normal configuration in at most $D + 1$ rounds. Then, from Lemma 4.11, the system enters a configuration where $\forall p :: p \in Tree(r)$ holds in at most $\lceil n / l \rceil \times (5 \times h(Tree(r)) + 2 \times l + 4$ rounds. From Lemma 4.10, the root sends its tokens and initiates a new PIF in at most $\lceil n / l \rceil \times (5 \times h(Tree(p)) + 2 \times l + 4$ rounds. So, we reach a configuration {EIN in at most $\lceil n / l\text{-}1 \rceil \times (5 \times h(Tree(r)) + 2 \times l + 4 + D + 1$ rounds. Since $h(Tree(r) \geq D$, the results follows.

Remark 4.1 $\gamma \in INC \Rightarrow (T_{net\gamma} = T_{potr} = l)$

By Lemmas 4.10, 4.11, and 4.12 the system will eventually reach a configuration $\in INC$. Then, the root sends a wave of l tokens and every token is

pushed towards the leaves. After all l tokens are distributed, the root initiates a next PIF. It is obvious that starting from any configuration $\in INC$, Algorithm 3.1 does not create any extra tokens.

Theorem 4.2 *(Safety)* $\forall_\alpha \in INC$: $\forall e \in \varepsilon_\alpha :: \forall \gamma \in e) \Rightarrow T_{net} \leq l$.

Since the root sends tokens infinitely often, by Macro Next, every node receives a token in a finite number of steps. The following theorem gives a bound on the number of waves a process in $Tree(r)$ must wait before it receives a token.

Theorem 4.3 *(Fairness) Starting in any arbitrary system configuration* $\gamma \in INC$, *each node* $p \in Tree(r)$ *will eventually receive (at least) one token during first W waves of tokens such that* $W \geq \lceil (\prod_{q \in \mu(rP_p)} (\#Child_q)) \mid l \rceil$ *if* $p \neq r$ *and* $W = 1$ *if* $p = r$.

Proof. We will prove this theorem by induction on length L_p *of* $\mu(rp)$.

Basic Step. The case where $p = r$ is obvious. Consider the case where $L_p = 1$, so $p \in N_r$. It is easy to see that if $Wl \geq \delta_r$ then each node p receives at least one token. We notice that if we choose W such that $Wl \geq m\ \delta_r$ (m is an integer such that $m \neq 0$), then p receives at least m tokens.

Induction step. Assume that the theorem is true for $L_p \leq k\ 1, k > 1$. Consider the case where $L_p = k$. By hypothesis, if $Wl \geq (\prod_{q \in \mu(rPpp)} (\#Child_q))$, P_p will receive at least one token, and if we choose W such that:

$Wl \geq (\#Child_{Pp})(\prod_{q \in \mu(rPp)})(\#Child_q) \Rightarrow W \geq \lceil \prod_{q \in \mu(rPp)} (\#Child_q))l \rceil$, then P_p receives at least $\#Child_{Pp}$ tokens. This will enable each of its children to receive at least one token, in particular the node p.

The following result follows from Theorems 4.2 and 4.3.

Theorem 4.4 *(Correctness) Starting in any arbitrary system configuration, the system satisfies the properties of safety and fairness.*

The following result follows from Lemma 4.12 and Theorem 4.4.

Theorem 4.5 *(Self-stabilization) Algorithm 3.1 is a self-stabilizing l-exclusion algorithm.*

State Complexity

One of the main parameters to measure the efficiency of self-stabilizing algorithms is the state complexity i.e., memory requirement per processor. In distributed systems, it is highly desired to propose algorithms those do not depend on the global properties (such as network size) which can be modified at any time. Therefore, we propose an algorithm whose space complexity is

independent of the size of the network n for any processor, and is independent of l for all processors except the root. Since, the space requirement is $(l + 3)\delta_r$ states (or $\lceil \log((l + 3)\delta_r \rceil)$ bits) for the root r and $4 \times \delta_p^2 \times D$ states (or $\lceil 4 \log((\delta_p \times D \rceil)$ bits) for each non root process p, where δ_p is the degree of process p and D is the diameter of the communication network. The root process r maintains two variables: $T_r \in \{0, 1, , l- 1, R,B,C\}$ and $S_r \in \{1,....,\delta_p\}$. The non root process p maintains four variables: $T_p \in \{T_{ok}, C,B,F\}$, S_p, $P_p \in \{1,...,\delta_p\}$, and L_p: 0,...,D. In the context of previous deterministic algorithms, the space complexity is $(l \times n^2)$ on rings, $O(2^n)$ on complete networks, and $O(Max^{2\delta+1})$ $(Max \geq l)$ on trees. However, the space requirement of our algorithm mapped on trees is $(l + 3)\delta_r$ states for the root r and $4 \times (\delta_p - 1)$ states for each non root process p, while it is only $(l + 1)$ states for the root r and 3 states for each non root process p if mapped on rings (Villain, 1999; Hadid, 2002). To our knowledge, since in general the diameter of a network is much smaller than the number of nodes, the space complexity of the presented algorithm over previously known approaches is efficiency. There remains the open question of state optimality for self-stabilizing l-exclusion algorithm.

Notes on Liveness

As stated before, a drawback of our algorithm, as in many deterministic self-stabilizing solutions to this problem in the current literature, is that we cannot ensure that every execution of our algorithm always satisfies the liveness property. On one hand, the root and any internal process can pass only one token at a time and among a sequence of three processors at most one token can exist. On other hand, the tokens that disappear at the leaves will not be redistributed at the root process during this distribution. These two properties go against increasing the number of potential privileged processors.

The evaluation of the number of processors having the tokens, and hence having the privilege to enter the critical section, simultaneously depend strictly on three parameters: the size of the network, the network topology, and l, which makes this task more complex. However, based on the assumption $l \leq \lceil n/3 \rceil$, we can observe that in any computation on various topologies, there exist some configurations where l processes hold privileges concurrently.

Unfair Deamon

In this section we present sketch a proof to show that our algorithm works also under an unfair daemon by showing that every process p will move infinitely often even if the daemon is unfair. The proof depends on the initial system configuration. If process p is in the tree rooted at the root process r. Then, as our algorithm behaves similarly to the algorithm presented in (Hadid, 2002) working under an unfair daemon, so process p moves infinitely often. Otherwise,

we need to show that process p will join the tree rooted at r. For that purpose, we can show by induction on the distance from p to r that process p will receive a token from r through its neighboring q in the tree rooted at r. Then, as long as process p is not selected by the daemon to receive the token from q, process q will not also be selected to participate in the next phase of the algorithm (i.e., PFC). However, as process q is in the tree so both processes will move. Then, process p will join the tree rooted at r, hence will move infinitely often.

CONCLUSIONS.

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. Our algorithm stabilizes in only O $(\lceil n / l \rceil \times (l + D))$ rounds. Its space requirement is $(l + 3)\delta_r$ states (or $\lceil \log ((l + 3)\delta_r \rceil)$ bits) for the root r and $4 \times \delta_p^2 \times$ D states (or $\lceil 4 \log (\delta_p \times D \rceil)$ bits) for a non-root process p. This is the first l-exclusion algorithm on arbitrary network in which the space requirement is independent of the size of the network for any process, and 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, is that we cannot ensure that every execution of our algorithm always satisfies the *liveness* property (low-level specification): 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. Observe that our algorithm allows at most one token to exist in a sequence of three processes. So, based on the assumption $l \leq \lceil n / 3 \rceil$, we can observe that in any computation on numerous 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

Abraham, U., Dolev, S., Herman, T., and Koll, I. (1997). Self-stabilizing l-exclusion. In Proceedings of the Third Workshop on Self-Stabilizing Systems, Carleton University Press, pages 48-63.

Abraham, U., Dolev, S., Herman, T., and Koll, I. (2001). Self-stabilizing l -exclusion. Theoretical Computer Science, 266:1-2:653-692.

Afek, Y., Dolev, D., Gafni, E., Merritt, M., and Shavit, N. (1990). A bounded rst-in, rst-enabled solution to the l-exclusion problem. Proceedings of the 4th International Workshop on Distributed Algorithms, Springer-Verlag, LNCS, 486:422-431.

Antonoiu, G. and Srimani, P. (2000). Self-stabilizing depth- rst multi-token circulation in tree networks. International Journal of Parallel, Emergent and Distributed Systems, 16:1:17-35.

Bernard, T., Bui, A., Flauzac, O., and Nolot, F. (2010). A multiple random walks based self-stabilizing k-exclusion algorithm in ad-hoc networks. International Journal of Parallel, Emergent and Distributed Systems, Taylor Francis eds, 25:2:135-152.

Cournier, A., Datta, A., Petit, F., and Villain, V. (2002). Self-stabilizing pif algorithm in arbitrary rooted networks. In 21st International Conference on Distributed Computing Systems, IEEE Computer Society Press, pages 91-98.

Datta, A., Devismes, S., Horn, F., and Larmore, L. (2011). Self-stabilizing k-out-of-l exclusion in tree networks. Int. J. Found. Comput. Sci., 22(3):657-677.

Datta, A., Hadid, R., and Villain, V. (2003a). A new self-stabilizing k-out-of- l exclusion algorithm on rings. In Self-Stabilizing Systems, ser. Lecture Notes in Computer Science, S.-T. Huang and T. Herman, Eds., 2704:113-128.

Datta, A., Hadid, R., and Villain, V. (2003b). A self-stabilizing token-based k-out-of-l exclusion algorithm. Concurrency and Computation: Practice and Experience, 15:1069 -1091.

Dijkstra, E. (1974). Self stabilizing systems in spite of distributed control. Communications of the Association of the Computing Machinery, 17:11:643{644.

Dolev, D., Gafni, E., and Shavit, N. (1988). Toward a non-atomic era: l -exclusion as test case. Proceeding of the 20th Annual ACM Symposium on Theory of Computing, Chicago, pages 78-92.

Dolev, S., Israeli, A., and Moran, S. (1997). Uniform dynamic self-stabilizing leader election. IEEE Transactions on Parallel and Distributed Systems, 8:4:420-440.

Fisher, M., Lynch, N., Burns, J., and Borondin, A. (1979). Resource allocation with immunity to limited process failure. Proceedings of the 20th IEEE Annual Symposium on Foundations of Computer Science, pages 234-254.

Fisher, M., Lynch, N., Burns, J., and Borondin, A. (1989). A distributed fifo allocation of identical resources using small shared space. ACM Transactions Programming Languages Systems, 11:90-144.

Flatebo, M., Datta, A., and Schoone, A. (1994). Self-stabilizing multi-token rings. Distributed Computing, Vol. 8, 8:133{142.

Gartner, F. (2003). A survey of self-stabilizing spanning-tree construction algorithms. School of Computer and Communication Sciences, Technical Report IC/2003/38.

Gouda, M. and Haddix, F. (1996). The stabilizing token ring in three bits. Journal of Parallel and Distributed Computing, 35:43-48.

Gradinariu, M. and Tixeuil, S. (2001). Tight space self-stabilizing uniform l-mutual exclusion. Distributed Computing, Vol. 7, pages 83-90. 32

Hadid, R. (2000). Space and time efficient self-stabilizing l -exclusion in tree networks. Proceedings of the 14th IEEE International Parallel and Distributed Processing Symposium, pages 529-534.

Hadid, R. (2002). Space and time efficient self-stabilizing l -exclusion in tree networks. Journal of Parallel and Distributed Computing, 62(5):843-864.

Hadid, R. and Villain, V. (2001). A new efficient tool for the design of self-stabilizing l -exclusion algorithms: the controller. In Proceedings of the 5th IEEE International Work-shop, WSS, pages 137-151.

Masum, S., Akbar, M., Ali, A., and Rahman, M. (2010). A consensus-based l -exclusion algorithm for mobile ad hoc networks. Ad Hoc Networks Journal, 8:30-45.

Peterson, G. (1990). Observation on l -exclusion. Proceedings of the 28th Annual Allerton Conference on Communication, Control and computing, Monticello, pages 568-577.

Villain, V. (1999). A key tool for optimality in the state model. In DIMACS'99, The 2nd Workshop on Distributed Data and Structures, Carleton University Press, pages 133-148.

Submitted : *06/03/2013*
Revised : *15/07/2013*
Accepted : *03/12/2013*

Journal of Engg. Research **Vol. 2 - (1) March 2014** pp. 95-120

خوارزمية الـ***l*-excleusion** في شبكة المتجذرة

***راشد حديد و **مهمت هاكان كراتا**

* قسم هندسة الكمبيوتر والعمارة، جامعة أيدين، بسيول ماه، ادوندو كاد، رقم: 38
سيفاكوف - كعسكسينيسي/ أسطنبول، تركيا

** قسم علوم الكمبيوتر وقسم هندسة الكمبيوتر - صندوق بريد 5969، صفاة، 13060 الكويت

خلاصة

في هذا البحث، نقدم أول حل مستقر لمشكلة الـ***L*-exclusion** في الشبكات. هذه المشكلة هي صورة معممة لمشكلة الـ mutual exclusion لعدد ($l \geq 1$) l عملية، حيث أن l عملية لديها الإمكانية لاستخدام الموارد المشتركة آنياً. الخوارزمية المستخدمة شبه منتظمة، ومتطلب المساحة لها هو δ_r ($l+3$) حالة (أو $\lceil \log((l+3))\Delta_r \rceil$) بت للجذر r و $\times_p^2 \times D_4$ حالة (أو $\lceil 4\log(\delta_p \times D \rceil$ هي δ_p، حيث أن p بت) لباقي العمليات غير الجذر (خوارزمية على الشبكات بخاصة l-exclusion هي قطر شبكة الاتصال. هذه أول D، وحيث أن p درجة العملية لجميع العمليات باستثناء الجذر. l أن المساحة المطلوبة من أي عملية لا تعتمد على حجم الشبكة، ولا تعتمد على البروتوكول المقترح موزع، حتمي، ولا يستخدم الشجرة الممتدة سابقة التكوين. وحيث أن الخوارزمية متزنة، فهي لا n جولة، حيث أن $O(\lceil n/l \rceil \times (l+D))$ تحتاج لتهيئة ابتدائية وتقاوم الخلل العابر. استقرار الخوارزمية يحتاج لـ هو حجم الشبكة.