A Self-stabilizing Token-Based k-out-of-ℓ Exclusion Algorithm

Ajoy K. Datta[1], Rachid Hadid*[2], and Vincent Villain[2]

[1] Department of Computer Science, University of Nevada, Las Vegas
[2] LaRIA, Université de Picardie Jules Verne, France

Abstract. In this paper, we present the first self-stabilizing solution to the k out of ℓ exclusion problem [14] on a ring. The k out of ℓ exclusion problem is a generalization of the well-known mutual exclusion problem — there are ℓ units of the shared resources, any process can request some number k $(1 \leq k \leq \ell)$ of units of the shared resources, and no resource unit is allocated to more than one process at one time. The space requirement of the proposed algorithm is independent of ℓ for all processors except a special processor, called Root. The stabilization time of the algorithm is only $5n$, where n is the size of the ring.

Keywords: Fault-tolerance, k-out-of-ℓ exclusion, mutual exclusion, resource allocation, self-stabilization.

1 Introduction

Fault-tolerance is one of the most important requirements of modern distributed systems. Various types of faults are likely to occur at various parts of the system. The distributed systems go through the transient faults because they are exposed to constant change of their environment. The concept of self-stabilization [7] is the most general technique to design a system to tolerate arbitrary transient faults. A self-stabilizing system, regardless of the initial states of the processors and initial messages in the links, is guaranteed to converge to the intended behavior in finite time. In 1974, Dijkstra introduced the property of self-stabilization in distributed systems and applied it to algorithms for mutual exclusion.

The *ℓ-exclusion problem* is a generalization of the mutual exclusion problem–ℓ processors are now allowed to execute the critical section concurrently. This problem models the situation where there is a pool of ℓ units of a shared resource and each processor can request at most one unit. In the last few years, many self-stabilizing ℓ-exclusion algorithms have been proposed [2,8,9,10,18].

The *k-out-of-ℓ exclusion* approach allows every processor to request k $(1 \leq k \leq \ell)$ units of the shared resource concurrently, but, no unit is allocated to multiple processors at the same time [14]. One example of this type of resource sharing is the sharing of channel bandwidth: the bandwidth requirements vary among the requests multiplexing

* Supported in part by the Pôle de Modélisation de Picardie, France and the Fonds Social Européen.

B. Monien and R. Feldmann (Eds.): Euro-Par 2002, LNCS 2400, pp. 553–562.

on the channel. For example, the demand would be quite different for a video than an audio transmission request.

Algorithms for k-out-of-ℓ exclusion were given in [3,12,13,14,15]. All these algorithms are permission-based: a processor can access the resource after receiving a permission from all the processors of the system [14,15] or from the processors constituting the quorum it belongs to [12,13].

Contributions. In this paper, we present the first self-stabilizing protocol for the k-out-of-ℓ exclusion problem. Our algorithm works on uni-directional rings and is *token-based*: a processor can enter its critical section, i.e., access the requested (k) units of the shared resource only upon receipt of k tokens. The space requirement of our algorithm is independent of ℓ for all processors except Root. The stabilization time of the protocol is only $5n$, where n is the size of the ring.

Outline of the Paper. In Section 2, we describe the model used in this paper, and present the specification of the problem solved. We propose a self-stabilizing k-out-of-ℓ exclusion protocol on rings in Section 3[1]. Finally, we make some concluding remarks in Section 4.

2 Preliminaries

2.1 The Model

The distributed system we consider in this paper is a uni-directional ring. It consists of a set of processors denoted by 0,1,..,n-1 communicating asynchronously by exchanging messages. Processors are anonymous. The subscripts 0,1,...,n-1 for the processors are used for the presentation only. We assume the existence of a distinguished processor (Processor 0), called Root. Each processor can distinguish its two neighbors: the left neighbor from which it can receive messages and the right neighbor it can send messages to. The left and right neighbors of Processor i are denoted by $i-1$ and $i+1$, respectively, where indices are taken modulo n. We assume that the message delivery time is finite but unbounded. We also consider a message to be in transit until it is processed by the receiving processor. Moreover, each link is assumed to be of bounded capacity, FIFO, and reliable (the messages are neither lost nor corrupted) during and after the stabilization phase. Our protocols are *semi-uniform* as defined in [6] — every processor with the same degree executes the same program, except one processor, Root. The messages are of the following form: $< message\text{-}type, message\text{-}value >$. The $message\text{-}value$ field is omitted if the message does not carry any value. Some messages contain more than one $message\text{-}value$. The program consists of a collection of actions. An action is of the form: $<guard> \longrightarrow < statement>$. A *guard* is a boolean expression over the variables of the processor and/or an *input* message. A *statement* is a sequence of assignments and/or message sending. An action can be executed only if its guard evaluates to true. We assume that the actions are atomically executed, meaning that the evaluation of a guard and the execution of the corresponding statement of an action, if executed, are done in one atomic step. The atomic execution of an action of p is called a *step* of p. When several actions of a processor are simultaneously enabled, then only the first enabled

[1] Due to space limitations, the proof of correctness is omitted. See [5] for the proofs

action (as per the text of the protocol) is executed. The *state* of a processor is defined by the values of its variables. The *state* of a system is a vector of n+1 components where the first n components represent the state of n processors, and the last one refers to the multi-set of messages in transit in the links. We refer to the state of a processor and the system as a (*local*) *state* and *configuration*, respectively.

2.2 Self-stabilization

Definition 1 (Self-stabilization). *A protocol $\mathcal{P}$ is self-stabilizing for a specification $\mathcal{SP}$ (a predicate defined over the computations) if and only if every execution starting from an arbitrary configuration will eventually reach (*convergence*) a configuration from which it satisfies $\mathcal{SP}$ forever (*closure*).*

In practice, we associate to $\mathcal{P}$ a predicate $\mathcal{L}_{\mathcal{P}}$ (called the *legitimacy predicate*) on the system configurations. $\mathcal{L}_{\mathcal{P}}$ must satisfy the following property: Starting from a configuration α satisfying $\mathcal{L}_{\mathcal{P}}$, $\mathcal{P}$ always behaves according to $\mathcal{SP}$, and any configuration reachable from α satisfies $\mathcal{L}_{\mathcal{P}}$ (closure property). Moreover if any execution of $\mathcal{P}$ starting from an arbitrary configuration eventually reaches a configuration satisfying $\mathcal{L}_{\mathcal{P}}$ (convergence property), we say that $\mathcal{P}$ stabilizes for $\mathcal{L}_{\mathcal{P}}$ (hence for $\mathcal{SP}$). The worst delay to reach a configuration satisfying $\mathcal{L}_{\mathcal{P}}$ is called the *stabilization time*.

2.3 The k-out-of-ℓ Exclusion Problem

In this section, we present the specification of the (k, ℓ)-exclusion problem. We will define the usual properties: *safety* and *fairness*. We also need to add another performance metric, called (k, ℓ)-*liveness*. An algorithm satisfying this property attempts to allow several processors to execute their critical section simultaneously. In order to formally define this property and get the proper meaning of the property, we assume that a processor can stay in the critical section forever. Note that we make this assumption only to define this property. Our algorithm does assume that the critical sections are finite. Informally, satisfying the (k, ℓ)-liveness means that even if some processors are executing their critical section for a long time, eventually some requesting processors can enter the critical section provided the safety and fairness properties are still preserved.

Definition 2. (k, ℓ)**-Exclusion Specification**

1. Safety: *Any resource unit can be used by at most one process at one time.*

2. Liveness: *(a)* Fairness: *Every request is eventually satisfied.*

(b) (k, ℓ)-liveness: *Let I be the set of processors executing their critical section forever, and every processor $i \in I$ using k_i units of the shared resource such that $\sum_{i \in I} k_i < \ell$. Let $\alpha = \ell - \sum_{i \in I} k_i$. Let J be the set of processors requesting the entry to their critical section such that every processor $j \in J$ needs $k_j \leq \alpha$ units of the resource. Then some of the processors in J will be eventually granted entry to the critical section provided they maintain the* safety *and* fairness *properties.*

Note that fairness and (k, ℓ)-liveness properties would not be related with each other if we did not include the fairness property in the (k, ℓ)-liveness property. On one hand, a classical mutual exclusion protocol can be a solution of the (k, ℓ)-exclusion problem

which does not satisfy the (k, ℓ)-liveness property. On the other hand, it is easy to design a protocol that always allows a processor in J (as defined in (k, ℓ)-liveness property) to enter the critical section. However, if the set J remains non-empty forever, then a processor requesting more than α units (hence not in J) may never get a chance to enter the critical section (starvation).

In uni-directional rings, we can use a token-based algorithm to maintain an ordering among the requests by circulating the tokens in a consistent direction. Then this solution would guarantee both fairness and (k, ℓ)-liveness properties.

In the k-out-of exclusion problem, if the maximum number of units (denoted as K) any process can request to access the critical section is known, then the space requirement depends only on K. Obviously, $K \leq \ell$.

A k-out-of-ℓ exclusion algorithm is self-stabilizing if every computation starting from an arbitrary initial configuration, eventually satisfies the safety, fairness, and (k, ℓ)-*liveness* requirements.

2.4 Parametric Composition

The parametric composition of protocols P_1 and P_2 was first presented in [10]. This is a generalization of the *collateral composition* of [16] and *conditional composition* of [4]. It allows both protocols to read the variables written by the other protocol. This scheme also allows the protocols to use the predicates defined in the other protocol. Informally, P_1 can be seen as a tool used by P_2, where P_2 calls some "public" functions of P_1 (we use the term *function* here with a generic meaning: it can be the variables used in the collateral composition or the predicates as in the conditional composition), and P_1 can also use some functions of P_2 via the "parameters".

Definition 3 (Parametric composition). *Let P_1 be a protocol with a set of* parameters *and a* public *part. Let P_2 be a protocol such that P_2 uses P_1 as an "external protocol". P_2 allows P_1 to use some of its functions (function may return no result) by using the parameters defined in P_2. P_1 allows protocol P_2 to call some of its functions by using the public part defined in P_1. The* parametric composition *of P_1 and P_2, denoted as $P_1 \triangleright_{\mathsf{P}} P_2$, is a protocol that has all the variables and all the actions of P_1 and P_2.*

The implementation scheme of P_1 and P_2 is given in Algorithm 1. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be predicates over the variables of P_1 and P_2, respectively. We now define a *fair* composition w.r.t. both protocols and define what it means for a parametrically composed algorithm to be self-stabilizing.

Definition 4 (Fair execution). *An execution e of $P_1 \triangleright_{\mathsf{P}} P_2$ is fair w.r.t. P_i $(i \in \{1, 2\})$ if either e is finite, or e contains infinitely many steps of P_i, or contains an infinite suffix in which no step of P_i is enabled.*

Definition 5 (Fair composition). *$P_1 \triangleright_{\mathsf{P}} P_2$ is fair w.r.t. P_i $(i \in \{1, 2\})$ if any execution of $P_1 \triangleright_{\mathsf{P}} P_2$ is fair w.r.t. P_i.*

The following composition theorem and its corollary are obvious:

Theorem 1. *If the composition $P_1 \triangleright_{\mathsf{P}} P_2$ is fair w.r.t. P_1, is fair w.r.t. P_2 if P_1 is stabilized for $\mathcal{L}_1$, protocol P_1 stabilizes for $\mathcal{L}_1$ even if P_2 is not stabilized for $\mathcal{L}_2$, and protocol P_2 stabilizes for $\mathcal{L}_2$ if $\mathcal{L}_1$ is satisfied, then $P_1 \triangleright_{\mathsf{P}} P_2$ stabilizes for $\mathcal{L}_1 \wedge \mathcal{L}_2$.*

Algorithm 1. $P_1 \triangleright_{\mathsf{P}} P_2$

Protocol $\mathbf{P}_1$($\mathbf{F'}_1 : TF_1, \mathbf{F'}_2 : TF_2, ..., \mathbf{F'}_\alpha : TF_\alpha$);	**Protocol** $\mathbf{P}_2$
Public	**External Protocol** $\mathbf{P}_1$($\mathbf{F}_1 : TF_1, \mathbf{F}_2 : TF_2$,...,$\mathbf{F}_\alpha : TF_\alpha$);
$\mathbf{Pub}_1 : TP_1$	**Parameters**
/* definition of Function $\mathbf{Pub}_1$ */	$\mathbf{F}_1 : TF_1$
...	/* definition of Function $\mathbf{F}_1$ */
$\mathbf{Pub}_\beta : TP_\beta$	...
/* definition of Function $\mathbf{Pub}_\beta$ */	$\mathbf{F}_\alpha : TF_\alpha$
	/* definition of Function $\mathbf{F}_\alpha$ */
begin	**begin**
...	...
[] $< Guard > \longrightarrow < statement >$	[] $< Guard > \longrightarrow < statement >$
/* Functions $\mathbf{F'}_i$ can be used	/* Functions $\mathbf{P}_1.\mathbf{Pub}_i$ can be used
in Guards and/or statements */	in Guards and/or statements */
...	...
end	**end**

Corollary 1. *Let $P_1 \triangleright_{\mathsf{P}} P_2$ be a self-stabilizing protocol. If Protocol P_1 stabilizes in t_1 for $\mathcal{L}_1$ even if P_2 is not stabilized for $\mathcal{L}_2$ and Protocol P_2 stabilizes in t_2 for $\mathcal{L}_2$ after P_1 is stabilized for $\mathcal{L}_1$, then $P_1 \triangleright_{\mathsf{P}} P_2$ stabilizes for $\mathcal{L}_1 \wedge \mathcal{L}_2$ in $t_1 + t_2$.*

3 Self-stabilizing k-out-of-ℓ Exclusion Protocol

The protocol presented in this section is token-based, meaning that a requesting processor receiving k ($1 \leq k \leq \ell$) tokens can enter the critical section. The protocol is based on a couple of basic ideas. First, we need a scheme to circulate ℓ tokens in the ring such that a processor cannot keep more than k tokens while it is in the critical section. Second, we use a method to make sure that any requesting processor eventually obtains the requested tokens. We use the parametric composition of two protocols: *Controller* (see Algorithm 2) and *ℓ-Token-Circulation* (see Algorithms 3 and 4), denoted as *Controller* $\triangleright_{\mathsf{P}}$*$\ell$-Token-Circulation*. We describe these two protocols next.

Controller. The protocol **Controller** (presented in Algorithm 2) implements several useful functions in the process of designing the k-out-of-ℓ exclusion algorithm. The controller keeps track of the number of tokens in the system. If this number is less

Algorithm 2. Controller.

For Root	**For Other Processors**
Controller(START, COUNT-ET)	**Controller**(T-ENABLED)
Variables MySeq: 0..MaxSeq	**Variables** MySeq: 0..MaxSeq
(A_{c1}) [] (**receive** $<CToken, Seq>$) $\wedge (MySeq = Seq) \longrightarrow$ $MySeq := MySeq + 1$ START **send** $<CToken, MySeq>$	(A_{c4}) [] (**receive** $<CToken, Seq>) \longrightarrow$ **if** $(MySeq \neq Seq)$ **then** $MySeq := Seq$ **for** $t = 1$ **to** T-ENABLED **do** **send** $<Enabled\_Token>$ **send**$<CToken, MySeq>$
(A_{c2}) [] **receive** $<Enabled\_Token> \longrightarrow$ COUNT-ET	(A_{c5}) [] **receive** $<Enabled\_Token> \longrightarrow$ **send** $<Enabled\_Token>$
(A_{c3}) [] **timeout** $\longrightarrow$ **send** $<CToken, MySeq>$	

(more) than ℓ, it replenishes (resp., destroys) tokens to maintain the right number (ℓ) of tokens in the system. The main burden of the above tasks (of the controller) is taken by Root. Root maintains two special variables C_e and C_f to implement these tasks (in Algorithm 3). We should point out here that these two variables are maintained only at Root. The detailed use of these variables and the implementation of the controller are explained below.

Root periodically initiates a status checking process by sending a special token, called $CToken$ (Actions A_{c1} and A_{c3} of Algorithm of 3). (Note that we refer to the token used by the controller as $CToken$ to distinguish it from the tokens used by the k-out-of-ℓ exclusion algorithm.) The $CToken$ circulation scheme is similar to the ones in [1,17]. Every time Root initiates a $CToken$, it uses a new sequence number ($MySeq$) in the $< CToken, Seq >$ message (Actions A_{c1} and A_{c3}). Other processors use one variable ($MySeq$) to store the old and new sequence numbers from the received $CToken$ messages (Actions A_{c4}). Now, we describe the maintenance of C_e and C_f at Root. Variable C_e records the number of "enabled tokens" in the system. Processors maintain two variables T_h and T_d in Algorithm 3. T_h indicates the number of tokens received (or originally held) by a processor. But, if a processor i is waiting to enter the critical section, i may be forced to "disable" some of these originally held active tokens. (We will describe this process in detail in the next paragraph.) T_d represents the number of disabled tokens. The disabled tokens cannot be used by a processor to enter the critical section until they are enabled later. The difference between T_h and T_d is what we call the "enabled tokens" in a processor. This is computed by Function `T-ENABLED` in Algorithm 3. On receiving a $CToken$ message from Root, a processor i computes the number of enabled tokens at i and then sends the same number of $<Enabled\_Token>$ message to its right number (Action A_{c4}). These $<Enabled\_Token>$ messages are forwarded by using Action A_{c5}. $<Enabled\_Token>$ messages eventually arrive at Root which then calculates the value of C_e (Action A_{c2}). Upon entering or exiting the critical section, processors send the extra enabled tokens (by using $<Token>$ message) to their right neighbor. As these $<Token>$ messages traverse the ring, the processors either use them (if needed) or forward to their right neighbor. The total number of these "free" enabled tokens are saved in C_f at Root. (See Algorithm 3 and 4 for details.)

Self-stabilizing ℓ-Token-Circulation. We briefly describe the interface between the ℓ-exclusion protocol and application program invoking the k-out-of-ℓ exclusion protocol. The interface comprises of three functions as described below:

1. Function STATE returns a value in $\{Request, In, Out\}$. The three values $Request$, In, and Out represent three modes of the application program "requesting to enter", "inside", and "outside" the critical section, respectively.

2. Function NEED returns the number of resource units (i.e., the tokens) requested by a processor.

3. Function ECS does not return a value. This function is invoked by the ℓ-exclusion protocol to send the permission to the application process to enter the critical section.

Algorithm 3. ℓ-Token-Circulation (Header).

For Root
ℓ-**Exclusion**(STATE() : $\{Request, In, Out\}$,
NEED(): $0..k$ $(k \leq K \leq \ell)$, ECS())
External Controller(START(),COUNT-ET())
Parameters
Function START()
00: **if** $(C_e + C_f + MyT_c + MyT_a) > \ell$ **then**
01: **send** $<Disable>$
02: **else for** $t = 1$ **to** $(\ell-(C_e+C_f+(MyT_c+MyT_a))$ **do**
03: **send** $<Token>$
04: **if** $(MyT_c + MyT_a > 0)$ **then**
05: **send** $<Allocate, MyT_c + MyT_a, MyOrder>$
06: **send** $<Collect, 0>$
07: $C_e := T_h - T_d$; $C_f := 0$
08: $MyT_c := 0$; $MyT_a := 0$
09: **end Function**
Function COUNT-ET()
if $C_f < \ell$ **then** $C_f := C_f + 1$
end Function
–
Function LOCK(): Boolean
Return(STATE() $= Request \wedge (0 < T_h <$ NEED))
end Function
Variables $T_h, T_d : 0..K$ $(K \leq \ell)$
C_e, C_f: $0..\ell$
MyT_c: $0..MaxV_c$ $(K \leq MaxV_c \leq \ell)$
MyT_a: $0..Min(2 \times MaxV_c, \ell)$
$MyOrder$: Boolean

For Other Processors
ℓ-**Exclusion**(STATE() : $\{Request, In, Out\}$,
NEED(): $0..k$ $(k \leq K \leq \ell)$, ECS())
External Controller(T-ENABLED())
Parameters
Function T-ENABLED(): Integer
$Return(T_h - T_d)$
end Function
–
Function LOCK(): Boolean
Return(STATE $= Request \wedge$
$(0 < T_h <$ NEED))
end Function
Variables $T_h, T_d : 0..K$ $(K \leq \ell)$
MyT_c: $0..MaxV_c$ $(K \leq MaxV_c \leq \ell)$
MyT_a: $0..Min(2 \times MaxV_c, \ell)$
$MyOrder$: Boolean

The basic objective of the algorithm in this section is to implement a circulation of ℓ tokens around the ring. A processor requesting k units of the resource can enter the critical section upon receipt of k tokens. The obvious approach to implement this would be the following: A requesting processor holds on to the tokens it receives until it gets the requested number (k) of tokens. When it receives k tokens, it enters the critical section. Upon completion of the critical section execution, it releases all the k tokens by sending them out to the next processor in the ring. Unfortunately, the above hold-and-wait approach is prone to deadlocks. Let α be the number of the (critical section entry) requesting processors in the system and β the total number of tokens requested by α processors. If $\beta \geq \ell + \alpha$, then ℓ tokens can be allocated in such a manner that every requesting processor is waiting for at least one token. So, the system has reached a deadlock configuration. We solve the deadlock problem by pre-emptying tokens. The method works in two phases as follows: 1. At least K tokens are disabled by pre-empting tokens from some processors. (Note that by definition, $k \leq K \leq \ell$.)

2. The disabled tokens are then used to satisfy the request of both the first waiting processor (w.r.t. the token circulation) with disabled tokens and the privileged[2] processor (say i). Processor i then enters the critical section. In order to ensure both fairness and (k, ℓ)-liveness, we construct a fair order in the ring (w.r.t. the token circulation) as follows: Every processor maintains a binary variable $MyOrder$ ($MyOrder \in \{true, false\}$). In Algorithms 3 and 4, two messages are used to implement the above two phases: $<Collect, T_c>$ and $<Allocate, T_a, Order>$ message. Root initiates both messages in

[2] The privileged processor is the first processor (w.r.t. the token circulation) whose $MyOrder$ is equal to that of Root. If all processors have their $MyOrder$ equal, then the privileged processor is Root.

Algorithm 4. ℓ-Token-Circulation (Actions).

For Root

(A_{l1}) [] STATE $\in \{Request, Out\} \wedge$
$(T_h + T_d > 0) \longrightarrow$
if STATE $= Out$ **then**
for $k = 1$ **to** $T_h - T_d$ **do**
send $<Token>$
$T_d := 0; T_h := 0$
else if $(T_h - T_d \geq$ NEED$)$ **then** ECS

(A_{l2}) [] (**receive** $<Token>$) $\longrightarrow$
if $(C_f + C_e < \ell)$ **then**
$C_e := C_e + 1$
if (STATE $\in \{Out, In\}$) **then**
send $< Token >$
else if $T_h < NEED$ **then** $T_h := T_h + 1$
else $T_d := T_d - 1$

(A_{l3}) [] ((**receive** $<Allocate, T_a, Order>$)
$\wedge (MyOrder = Order)) \longrightarrow$
$MyT_a := T_a$
if (STATE $= Request$) **then**
if $(MyT_a \geq NEED - (T_h - T_d))$ **then**
$MyT_a := MyT_a - (NEED - (T_h - T_d))$
$T_h := NEED ; T_d := 0$
$MyOrder := \overline{MyOrder}$
else if $(T_d \geq MyT_a)$ **then**
$T_d := T_d - MyT_a$
else $T_h := T_h + (MyT_a - T_d)$
$T_d := 0$
$MyT_a := 0$
else $MyOrder := \overline{MyOrder}$

(A_{l4}) [] (**receive** $<Allocate, T_a, Order>$) $\longrightarrow$
$MyT_a := T_a$
if $(T_d > 0 \wedge$ LOCK$)$ **then**
if $(MyT_a \geq NEED - (T_h - T_d))$ **then**
$MyT_a := MyT_a - (NEED - (T_h - T_d))$
$T_h := NEED ; T_d := 0$

(A_{l5}) [] (**receive** $<Collect, T_c>$)) $\longrightarrow$
$MyT_c := T_c$
if LOCK **then**
$MyT_c := Min(MyT_c + (T_h - T_d), MaxV_c)$
$T_d := T_d + (MyT_c - T_c)$

(A_{l6}) [] (**receive** $<Disable>$) $\longrightarrow$
$T_d := T_h$

For Other Processors

(A_{l7}) [] STATE $\in \{Request, Out\} \wedge$
$(T_h + T_d > 0) \longrightarrow$
if STATE $= Out$ **then**
for $k = 1$ **to** $T_h - T_d$ **do**
send $<Token>$
$T_d := 0; T_h := 0$
else if $(T_h - T_d \geq$ NEED$)$ **then** ECS

(A_{l8}) [] (**receive** $<Token>$) $\longrightarrow$
if (STATE $\in \{Out, In\}$) **then**
send $< Token >$
else if $T_h < NEED$ **then**
$T_h := T_h + 1$
else $T_d := T_d - 1$

(A_{l9}) [] ((**receive** $<Allocate, T_a, Order>$)
$\wedge (MyOrder \neq Order)) \longrightarrow$
$MyT_a := T_a$
if (STATE $= Request$) **then**
if $(MyT_a \geq NEED - (T_h - T_d))$ **then**
$MyT_a := MyT_a - (NEED - (T_h - T_d))$
$T_h := NEED ; T_d := 0$
$MyOrder := Order$
if $MyT_a > 0$ **then**
send $<Allocate, MyT_a, Order>$
else if $(T_d \geq MyT_a)$ **then**
$T_d := T_d - MyT_a$
else $T_h := T_h + (MyT_a - T_d)$
$T_d := 0$
$MyT_a := 0$
else $MyOrder := Order$
send $<Allocate, MyT_a, Order>$

(A_{l10}) [] (**receive** $<Allocate, T_a, Order>$) $\longrightarrow$
$MyT_a := T_a$
if $(T_d > 0 \wedge$ LOCK$)$ **then**
if $(MyT_a \geq NEED - (T_h) - T_d)$ **then**
$MyT_a := MyT_a - (NEED - (T_h - T_d))$
$T_h := NEED ; T_d := 0$
if $MyT_a > 0$ **then**
send $<Allocate, MyT_a>$
else send $<Allocate, MyT_a>$

(A_{l11}) [] (**receive** $<Collect, T_c>$)) $\longrightarrow$
$MyT_c := T_c$
if LOCK **then**
$MyT_c := Min(MyT_c + (T_h - T_d), MaxV_c)$
$T_d := T_d + (MyT_c - T_c)$
send $<Collect, MyT_c>$

(A_{l12}) [] (**receive** $<Disable>$) $\longrightarrow$
$T_d := T_h$; **send** $<Disable>$

Function START (Lines 05 and 06 of Algorithm 3). Root executes Function START before initiating a new $CToken$ message (see Algorithm 2).

The receipt of a $Collect$ message at a processor i has the following effect (see Actions A_{l5} and A_{l11} of Algorithms 4): If Processor i is waiting to enter the critical section (because it did not receive enough tokens yet) (verified by using Function LOCK), then the current enabled tokens at i are marked disabled and these tokens are added to the

pool of collected tokens in the $Collect$ message. Finally, i forwards the $Collect$ message to its right neighbor. The field T_c in $<Collect, T_c>$ message represents the number of disabled tokens collected so far from the processors in the ring. Every processor maintains a variable MyT_c corresponding to the message field T_c.

When Root receives the $Collect$ message back (Action A_{l5}), it stores the total number of disabled tokens (collected from all the other processors) in its own variable MyT_c. When a processor i receives an $<Allocate, T_a, Order>$ message, (the field $Order$ corresponds to $MyOrder$ of Root), i does the following (see Actions A_{l3}, A_{l4}, A_{l9}, and A_{l10}): If i is waiting to enter the critical section (i.e., i is requesting and contains at least one disabled token) or i is privileged (i.e., i is requesting and $MyOrder_i = Order$), then it will use some (or all) tokens from the pool of available tokens in the message field T_a. This would allow i to enter the critical section by executing Action A_{l7} (Action A_{l1} for Root). If there are some available tokens, i.e., T_a is not zero, i will pass on those tokens to its right neighbor by sending an $Allocate$ message. Thus, either Root receives an $Allocate$ message containing some left-over tokens, or all the available tokens are consumed by other processors. It should be noted that $Allocate$ message delivers its tokens (available in T_a) to a privileged processor i even if i's request cannot be granted (T_a is not enough) (see Actions A_{l3} and A_{l9}). But, if i is waiting, then $Allocate$ message delivers its tokens to i only if its request can be granted (T_a is enough) (see Actions A_{l4} and A_{l10}). As discussed earlier, Root maintains two special counters: C_e and C_f. The sum of C_e, C_f, MyT_c, and MyT_a represents the total number of tokens in the ring at the end of the $CToken$ traversal. If this number is more than ℓ, then Root destroys (or disables) all the tokens by sending a special message $<Disable>$ (Lines 00-01 and Actions A_{l6} and A_{l12}). But, if Root sees that there are some missing tokens in the ring, it replenishes them to maintain the right number (ℓ) of tokens in the system (Lines 03-04).

Proof Outline. The movement of the $CToken$ and enabled tokens are independent of each other except that they are synchronized at the beginning of a new traversal of the $CToken$ (Action A_{c1} and Function START). So, we can claim that the composed *Controller* $\triangleright_P$ ℓ-$Token$-$Circulation$ is fair w.r.t. *Controller*. We can borrow the result of [1,17] to claim that the $CToken$ stabilizes for the predicate "there exist only one $CToken$" in two $CToken$ traversal time, i.e., in $2n$. By the controller (Algorithm 2) (it maintains the right number ℓ of tokens in the system in at most three more $CToken$ traversal time) and the mechanism of pre-empting tokens, we can claim that deadlock cannot occur ($deadlock$-$freeness$). Moreover, it ensures the (k, ℓ)-liveness. By Algorithms 3 and 4 ($MyOrder$ construction), every processor i will be eventually privileged and i'request will eventually have higher priority than the rest of the requests in the system. Therefore, the composed *Controller* $\triangleright_P$ *ℓ-Token-Circulation* does not cause starvation of any processor. Then, our final result follows from Theorem 1 and Corollary 1: *Controller* $\triangleright_P$ *ℓ-Token-Circulation* stabilizes for the k-out-of-ℓ exclusion specification — safety, fairness, and (k, ℓ)-liveness — in at most five $CToken$ traversal time i.e., $5n$.

4 Conclusions

In this paper, we present the first self-stabilizing protocol for k-out-of-ℓ exclusion problem. We use a module called controller which can keep track of the the number of tokens

in the system by maintaining only a counter variable only at Root. One nice characteristic of our algorithm is that its space requirement is independent of ℓ for all processors except Root. The stabilization time of the protocol is $5n$. Our protocol works on uni-directional rings. However, we can use a self-stabilizing tree construction protocol and the Euler tour of the tree (virtual ring) to extend the algorithm for a general network.

References

1. Afek, Y., Brown, G.M : Self-stabilization over unreliable communication media. Distributed Computing, Vol. 7 (1993) 27–34
2. Abraham, U., Dolev, S., Herman, T., Koll, I. : Self-Stabilizing ℓ-exclusion. In Proceedings of the third Workshop on Self-Stabilizing Systems, International Informatics Series 7, Carleton University Press (1997) 48–63
3. Baldoni, R. : An $O(N^{M/M+1})$ distributed algorithm for the k-out of-M resources allocation problem. In Proceedings of the 14th conference on Distributed Computing and System (1994) 81-85.
4. Datta, AK., Gurumurthy, S., Petit, F., Villain V. : Self-stabilizing network orientation algorithms in arbitrary rooted networks. In Proceedings of the 20th IEEE International Conference on Distributed Computing Systems (2000) 576–583
5. Datta, AK., Hadid, R., Villain V. : A Self-stabilizing Token-Based k-out-of-ℓ Exclusion Algorithm. Technical Report RR 2002-04, LaRIA, University of Picardie Jules Verne (2002).
6. Dolev, D., Gafni, E., Shavit, N. : Toward a non-atomic era: ℓ-exclusion as test case. In Proceeding of the 20th Annual ACM Symposium on Theory of Computing, Chicago (1988) 78–92
7. Dijkstra, EW. : Self stabilizing systems in spite of distributed control. Communications of the Association of the Computing Machinery, Vol. 17, No. 11 (1974) 643–644
8. Flatebo, M., Datta, AK., Schoone, AA. : Self-stabilizing multi-token rings. Distributed Computing, Vol. 8 (1994) 133-142
9. Hadid, R. : Space and time efficient self-stabilizing ℓ-exclusion in tree networks. In Journal of parallel and distributed computing. To appear.
10. Hadid, R., Villain, V : A new efficient tool for the design of Self-stabilization ℓ-exclusion algorithms: the controller, In Proceedings of the 5th International Workshop, WSS (2001) 136-151.
11. Lamport , L. : Time, clocks, and the ordering of events in a distributed system. Communications of ACM, Vol. 21 (1978) 145-159
12. Manabe, Y., Tajima, N. : (h, k)-Arbiter for h-out of-k mutual exclusion problem. In Proceedings of the 19th Conference on Distributed Computing Systems, (1999) 216-223.
13. Manabe, Y., Baldoni, R., Raynal, M., Aoyagi, S.: k-Arbiter: A safe and general scheme for h-out of-k mutual exclusion. Theoretical Computer Science, Vol. 193 (1998) 97-112
14. Raynal, M. : A distributed algorithm for the k-out of-m resources allocations problem. In Proceedings of the 1st conference on Computing and Informations, Lecture Notes in Computer Science, Vol. 497 (1991) 599-609
15. Raynal, M. : Synchronisation et état global dans les systèmes répartis. Eyrolles, collection EDF (1992)
16. Tel, G. : Introduction to distributed algorithms. Cambridge University Press (1994)
17. Varghese G. : Self-stabilizing by counter flushing. Technical Report, Washington University (1993)
18. Villain V. A Key Tool for Optimality in the State Model. DIMACS Workshop on Distributed Data and Structures, Proceedings in Informatics 6, Carleton Scientific, pages 133-148, 1999.