International Journal of Foundations of Computer Science
© World Scientific Publishing Company

Self-Stabilizing k-out-of-ℓ Exclusion in Tree Networks

Ajoy K. Datta
School of Computer Science, University of Nevada
Las Vegas, USA
datta@cs.unlv.edu

Stéphane Devismes
VERIMAG, Université Joseph Fourier
Grenoble, France
stephane.devismes@imag.fr

Florian Horn
LIAFA, Université Paris Denis Diderot
Paris, France
florian.horn@liafa.jussieu.fr

Lawrence L. Larmore
School of Computer Science, University of Nevada
Las Vegas, USA
larmore@cs.unlv.edu

Received (Day Month Year)
Accepted (Day Month Year)
Communicated by (xxxxxxxxxx)

In this paper, we address the problem of k-out-of-ℓ exclusion, a generalization of the mutual exclusion problem, in which there are ℓ units of a shared resource, and any process can request up to k units ($1 \leq k \leq \ell$). A protocol is self-stabilizing if, starting from an arbitrary configuration, be it initial state or after a corrupted state, the protocol can start behaving normally within a finite time. We propose the first deterministic self-stabilizing distributed k-out-of-ℓ exclusion protocol in message-passing systems for asynchronous oriented tree networks which assumes bounded local memory for each process.

Keywords: Fault-tolerance; k-out-of-ℓ exclusion; oriented tree networks; resource allocation; self-stabilization.

1991 Mathematics Subject Classification: 68Q22, 68M15

1. Introduction

The basic problem in resource allocation is the management of shared resources, such as printers or shared variables. The use of such resources by one user affects their availability for other users. In those cases, at most one agent can access the resource at any time, using a special section of code called a *critical section.* The associated protocols must guarantee the *mutual exclusion* property: the critical section can be executed by at most one process at any time. The ℓ-exclusion property [5] is a generalization of mutual exclusion, where ℓ processes can execute the critical section simultaneously. Thus, in ℓ-exclusion, ℓ units of a same resource (*e.g.*, a pool of IP addresses) can be allocated. This problem can be generalized still further by considering heterogeneous requests, *e.g.*, bandwidth for audio or video streaming. The $\mathtt{k}$-out-of-ℓ exclusion property [11] allows us to deal with such requests; requests may vary from 1 to $\mathtt{k}$ units of a given resource, where $1 \leq \mathtt{k} \leq \ell$.

1.1. *Contributions*

In this paper, we propose a (deterministic) *self-stabilizing* distributed $\mathtt{k}$-out-of-ℓ exclusion protocol for asynchronous oriented tree networks. A protocol is self-stabilizing [3] if, after transient faults put the system into an illegitimate global state, the system recovers without external (*e.g.* human) intervention in finite time. Our protocol is written in the message-passing model, and assumes bounded memory per process. To the best of our knowledge, no protocol of this type has previously appeared in the literature.

Obtaining a self-stabilizing solution for the $\mathtt{k}$-out-of-ℓ exclusion problem in oriented trees is desirable, but also complex. Our main reason for dealing with oriented trees is that extension to general rooted networks is trivial; it consists of running the protocol concurrently with a spanning tree construction (for message passing systems) (see [4] for such algorithms). The complexity of the solution comes from the fact that the problem is a generalization of mutual exclusion.

Designing protocols for such problems on realistic systems often leads to obfuscated solutions. A direct consequence is then the difficulty of checking, or analyzing the solution. To circumvent this problem, we propose, here, a step-by-step approach. We start from a “naive” non-operating circulation of ℓ resource tokens. Incrementally, we augment this solution with several other types of tokens until we obtain a correct non fault-tolerant solution. We then introduce an additional control mechanism that guarantees self-stabilization assuming unbounded local memory. Finally, we modify the protocol to accommodate bounded local memory. We validate our approach by showing correctness of our protocol and analyzing its complexity.

1.2. *Related Work*

Two kinds of protocols are widely used in the literature to solve the $\mathtt{k}$-out-of-ℓ exclusion problem: permission-based protocols, and ℓ-token circulation. All non-self-stabilizing solutions currently in the literature are permission-based. In a permission-based protocol, any process can access a resource after receiving permission from all processes [11], or from the processes constituting its quorum [9, 10]. There exist two self-stabilizing solutions for $\mathtt{k}$-out-of-ℓ exclusion on the oriented rooted ring [1, 2]. These solutions are based on circulation of ℓ tokens, where each token corresponds to a resource unit.

1.3. *Outline*

The remainder of the paper is organized as follows: In Section 2, we define the computational model. We present our solution in Section 3. In Section 4, we provide the proof of correctness of our protocol. In Section 5, we analyze its complexity. We conclude in Section 6.

2. Preliminaries

2.1. *Distributed Systems*

We consider *asynchronous distributed systems* having a *finite* number of *processes*. Every process can directly communicate with a subset of processes called *neighbors*. We denote by Δ_p the number of neighbors of a process p. We consider the message-passing model where communication between neighboring processes is carried out by *messages* exchanged through *bidirectional links*, *i.e.*, each link can be seen as two *channels* in opposite directions. The neighbor relation defines a *network*. We assume that the topology of the network is that of an *oriented tree*. *Oriented* means that there is a distinguished process called *root* (denoted r) and that every non-root process knows which neighbor is its *parent* in the tree, *i.e.*, the neighbor that is nearest to the root.

A process is a sequential deterministic machine with input/output capabilities, bounded local memory, and that uses a local algorithm. Each process executes its local algorithm by taking *steps*. In a step, a process executes two actions in sequence: (1) either it tries to receive a message from another process, sends a message to another process, or does nothing; and then (2) modifies some of its variables.[a] The local algorithm is structured as in infinite loop that contains finitely many actions.

We assume that the channels incident to a process p are locally distinguished by a *label*, a number in the range $\{0 \dots \Delta_p - 1\}$; by an abuse of notation, we may refer to a neighbor q of p by the label of p's channel to q. For sake of simplicity, any non-root process locally numbers the channel to its parent by 0. We assume that the channels are *reliable*, meaning that no message can be lost[b] and *FIFO*, meaning that messages are received in the order they are sent. We also assume that each bidirectional link initially contains arbitrary messages, but not more than a given bound $\mathtt{C_{MAX}}$.[c]

A message is of the following form: $\langle type, value \rangle$. The *value* field is omitted if the message does not carry any value. A message may also contain more than one

[a] When there is ambiguity, we denote by x_p the variable x in the code of process p.

[b] *N.b.*, this assumption holds only after there are no more transient faults.

[c] This assumption is required to obtain a deterministic self-stabilizing solution working with bounded process memory; see [6].

value.

A *distributed protocol* is a collection of n local algorithms, one per process. We define the *state* of each process to be the state of its local memory and the contents of its incoming channels. The global state of the system, referred to as a *configuration*, is defined as the product of the states of processes. We denote by $\mathcal{C}$ the set of all possible configuration. An *execution* of a protocol $\mathcal{P}$ in a system $\mathcal{S}$ is an infinite sequence of configurations $\gamma_0\gamma_1 \dots \gamma_i \dots$ such that in any transition $\gamma_i \mapsto \gamma_{i+1}$ either a process take a step, or an external (*w.r.t.* the protocol) application modifies an input variable. Any execution is assumed to be *asynchronous* but *fair*: Every process takes an infinite number of steps in the execution but the time between two steps of a process is unbounded.

2.2. $\mathtt{k}$-*out-of-*$\boldsymbol{\ell}$ *Exclusion*

In $\mathtt{k}$-out-of-ℓ exclusion, the existence of ℓ units of a shared resource is assumed. Any process can request at most $\mathtt{k}$ units of the shared resource, where $\mathtt{k} \leq \ell$. We say that a protocol satisfies the $\mathtt{k}$-out-of-ℓ exclusion specification if it satisfies the following three properties:

- **Safety:** At any given time, each resource unit (*n.b.*, here a resource unit corresponds to a token) is used by at most one process, each process uses at most $\mathtt{k}$ resource units, and at most ℓ resource units are used.
- **Fairness:** If a process requests at most $\mathtt{k}$ resource units, then its request is eventually satisfied (*i.e.*, it can eventually use the resource unit it requests using a special section of code called *critical section*).
- **Efficiency:** Maximum possible number of requests must be satisfied simultaneously.

The notion of *efficiency* is difficult to define precisely. A more convenient parameter was introduced in [2] to formally characterize efficiency: $(\mathtt{k}, \ell)$-*liveness*. In [2], the concept of $(\mathtt{k}, \ell)$-liveness was introduced and justified by proving that without this property, the $\mathtt{k}$-out-of-$\boldsymbol{\ell}$ exclusion algorithms do not satisfy the fairness property. Assume that there is a subset I of processes such that every process in I is executing its critical section forever (*i.e.*, it holds some resource units forever). Let α be the total number of resource units held forever by the processes in I. Let R be the set of processes not in I that are requesting some resource units; for each $q \in R$, let r_q be the number of resource units being requested by q, and assume that $r_q \leq \ell - \alpha$ for all $q \in R$. Then, if $R \neq \emptyset$, at least one member of R eventually satisfies its request.

2.3. *Waiting Time*

If a process p requests the critical section, the *waiting time* is the number of times that processes other than p enter their critical sections before p enters its critical section.

2.4. *Interface*

In any k-out-of-ℓ exclusion protocol, a process needs to interact with the application that requests the resource units. To manage these interactions, we use the following interface at each process:

- State $\in \{Req, In, Out\}$. State $= Req$ means that the application is requesting some resource units. State switches from *Req* to *In* when the application is allowed access to the requested resource units. State switches from *In* to *Out* when the requested resource units are released into the system. The switching of State from *Req* to *In* and from *In* to *Out* is managed by the k-out-of-ℓ exclusion protocol itself, while the switching from *Out* to *Req* is managed by the application. Other transitions (for instance, *In* to *Req*) are forbidden.
- Need $\in \{0 \ldots \mathtt{k}\}$, the number of resource units currently being requested by the application.
- EnterCS(): *function.* This function is called by the protocol to allow the application to execute the *critical section.* From this call, the application has control of the resource units until the end of the critical section (we assume that the critical section is always executed in finite, yet unbounded, time).
- ReleaseCS(): *Boolean.* This predicate holds if and only if the application is not executing its critical section.

2.5. *Self-Stabilization*

A *specification* is a predicate over the set of all executions. A set of configurations $\mathcal{C}_1 \subseteq \mathcal{C}$ is an *attractor* for a set of configurations $\mathcal{C}_2 \subseteq \mathcal{C}$ if for any $\gamma \in \mathcal{C}_2$ and any execution whose initial configuration is γ, the execution contains a configuration of $\mathcal{C}_1$.

Definition 1 ([3]) *A protocol $\mathcal{P}$ is* self-stabilizing *for the specification SP in a system $\mathcal{S}$ if there exists a non-empty subset of $\mathcal{C}$, $\mathcal{L}$, such that: (i) Any execution of $\mathcal{P}$ in $\mathcal{S}$ starting from a configuration of $\mathcal{L}$ satisfies SP (*Closure Property*), and (ii) $\mathcal{L}$ is an attractor for $\mathcal{C}$ (*Convergence Property*).*

In the following, the configurations of $\mathcal{L}$ are called *legitimate.* All other configurations are said *illegitimate.*

We define *stabilization time* to be the maximum amount of time to reach a legitimate configuration, starting from an arbitrary configuration.

3. Protocol

In this section, we present our self-stabilizing k-out-of-ℓ exclusion protocol for oriented trees (Algorithms 1 and 2). Our solution uses circulation of several types of tokens. To clearly explain the function of these tokens, we adopt a step-by-step

Algorithm 1 k-out-of-ℓ exclusion on oriented trees, local algorithm for the root r

```
variables:
    $C$, myC $\in [0 \dots (n-1)(\texttt{C}_{\texttt{MAX}}+1)]$
    Succ $\in [0 \dots \Delta_r - 1]$
    RSet: multiset of at most k values in $[0 \dots \Delta_r - 1]$
    Need $\in [0 \dots \texttt{k}]$; State $\in \{Req, In, Out\}$
    Prio $\in \{\perp, 0, \dots, \Delta_r - 1\}$
    $R$, Reset: Boolean; SToken, $PT \in [0 \dots \ell + 1]$
    SPush, SPrio, $PPr \in [0 \dots 2]$
repeat forever
    for all $q \in [0 \dots \Delta_r - 1]$ do
        if (receive⟨ResT⟩ from $q$) $\wedge \neg$Reset then
            if (State $= Req$) $\wedge$ (|RSet| $<$ Need) then
                RSet $\leftarrow$ RSet $\cup \{q\}$
            else
                if $q = \Delta_r - 1$ then
                    SToken $\leftarrow \min(\texttt{SToken} + 1, \ell + 1)$
                end if
                send⟨ResT⟩ to $q + 1$
            end if
        end if
        if (receive⟨PushT⟩ from $q$) $\wedge \neg$Reset then
            if (Prio $\neq \perp$) $\wedge$ (State $\neq Req \vee$ |RSet| $<$ Need)$\wedge$
             (State $\neq In$) then
                for all $i \in$ RSet do
                    if $i = \Delta_r - 1$ then
                        SToken $\leftarrow \min(\texttt{SToken} + 1, \ell + 1)$
                    end if
                    send⟨ResT⟩ to $i + 1$
                end for
                RSet $\leftarrow \emptyset$
            end if
            if $q = \Delta_r - 1$ then
                SPush $\leftarrow \min(\texttt{SPush} + 1, 2)$
            end if
            send⟨PushT⟩ to $q + 1$
        end if
        if (receive⟨PrioT⟩ from $q$) $\wedge \neg$Reset then
            if Prio $= \perp$ then
                Prio $\leftarrow q$
            else
                send⟨PrioT⟩ to $q + 1$
            end if
        end if
        if (receive⟨ctrl, $C$, $R$, $PT$, $PPr$⟩ from $q$) then
            if ($q =$ Succ) $\wedge$ (myC $= C$) then
                Succ $\leftarrow$ Succ $+ 1$
                if Succ $= 0$ then
                    myC $\leftarrow$ myC $+ 1$
                    Reset $\leftarrow (PT + \texttt{SToken} > \ell) \vee$
                      $(PPr + \texttt{SPrio} > 1) \vee (\texttt{SPush} > 1)$
                    if Reset then
                        RSet $\leftarrow \emptyset$
                        Prio $\leftarrow \perp$
                    else
                        if $PPr + \texttt{SPrio} < 1$ then
                            send⟨PrioT⟩ to 0
                        end if
                        while $PT + \texttt{SToken} < \ell$ do
                            send⟨ResT⟩ to 0
                            SToken $\leftarrow \min(\texttt{SToken} + 1, \ell + 1)$
                        end while
                        if SPush $< 1$ then
                            send⟨PushT⟩ to 0
                        end if
                    end if
                    SToken $\leftarrow 0$
                    SPrio $\leftarrow 0$
                    SPush $\leftarrow 0$
                    $PT \leftarrow 0$
                    $PPr \leftarrow 0$
                end if
                $PT \leftarrow \min(PT + |\texttt{RSet}|_q, \ell + 1)$
                if Prio $= q$ then
                    $PPr \leftarrow \min(PPr + 1, 2)$
                end if
                send⟨ctrl, myC, Reset, $PT$, $PPr$⟩ to Succ
                RestartTimer()
            end if
        end if
    end for
    if (State $= Req$) $\wedge$ (|RSet| $\geq$ Need) then
        State $\leftarrow In$
        EnterCS()
    end if
    if (State $= In$) $\wedge$ ReleaseCS() then
        for all $i \in$ RSet do
            if $i = \Delta_r - 1$ then
                SToken $\leftarrow \min(\texttt{SToken} + 1, \ell + 1)$
            end if
            send⟨ResT⟩ to $i + 1$
        end for
        RSet $\leftarrow \emptyset$
        State $\leftarrow Out$
    end if
    if (Prio $\neq \perp$) $\wedge$ (State $\neq Req \vee$ |RSet| $\geq$ Need) then
        if Prio $= \Delta_r - 1$ then
            SPrio $\leftarrow \min(\texttt{SPrio} + 1, 2)$
        end if
        send⟨PrioT⟩ to Prio $+ 1$
        Prio $\leftarrow \perp$
    end if
    if TimeOut() then
        send⟨ctrl, myC, Reset, 0, 0⟩ to Succ
        RestartTimer()
    end if
end repeat
```

approach: we start from "naive" non-operating circulation of ℓ resource tokens. Incrementally, we augment this solution with several other types of tokens, until we obtain a non-fault-tolerant solution. We then add an additional control mechanism that guarantees self-stabilization, assuming unbounded local memory of processes. Finally, we modify our protocol to work with bounded memory.

3.1. *A Non-Fault-Tolerant Protocol*

The basic principle of our protocol is to use ℓ circulating *resource tokens* (the ResT messages) following depth-first search (DFS) order: when a process p receives a token from channel number i, and if that token is retransmitted, either immediately or later, it will be sent to its neighbor along channel number $i+1$ (modulo Δ_p). (This same rule will also be followed by all the types of tokens we will later describe.) Figure 1 shows the path followed by a token during depth-first circulation in an oriented tree (recall that any non-root process locally numbers the channel to its parent by 0). In this way, the oriented tree emulates a ring with a designated leader (see Figure 4), and we refer to the path followed by the tokens as the *virtual ring*.

As explained Section 2, the requests are managed using the variables `State` and `Need`. Each process also uses the multiset[d] variable `RSet` to collect the tokens; the collected tokens are said to be "reserved." While `State` $=$ *Req* and $|$`RSet`$| <$ `Need`, a process collects all tokens it receives; it also stores in `RSet` the number of the channel from which it receives each token, so that when it is finally retransmitted, it will continue its correct path around the virtual ring. When `State` $=$ *Req* and $|$`RSet`$| \geq$ `Need`, it enters the critical section: `State` is set to *In* and the function `EnterCS`() is called. Once the critical section is done (*i.e.*, when `State` $=$ *In* and the predicate `ReleaseCS`() holds) `State` is set to *Out*, all tokens in `RSet` are retransmitted, and `RSet` is set to $\emptyset$. When a process receives a token it does not need, either because it has enough to enter its critical section or because `State` $=$ *Out*, it immediately retransmits that token.

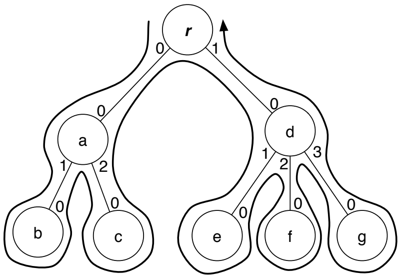


Fig. 1. Depth-first token circulation on oriented trees.

Unfortunately, such a simple protocol does not always guarantee liveness; Figure 2 shows a case where liveness is not maintained. In the example shown, there are five resources tokens (*i.e.*, $\ell = 5$) and each process can request up to three tokens (*i.e.*, $\mathtt{k} = 3$ for each process). In Configuration (i), processes a, b, c, and d request more tokens than they will receive. This configuration will lead to deadlock, namely Configuration (ii): processes a, b, c, and d reserve all the tokens they receive and never release them because their requests are never satisfied.

We can prevent deadlock by adding a new type of token, called the *pusher* (the message `PushT`). If the system is in a legitimate state, there is exactly one pusher. It permanently circulates through the virtual ring, and prevents a process that

[d] *N.b.* a multiset can contain several identical items.

Algorithm 2 k-out-of-ℓ exclusion on oriented trees, local algorithm for the other process p

```
variables:
    $C$, myC $\in [0 \ldots (n-1)(\mathtt{C_{MAX}}+1)]$
    Succ $\in [0 \ldots \Delta_p - 1]$
    RSet: multiset of at most k values in $[0 \ldots \Delta_p - 1]$
    Need $\in$ [0 . . . k]; State $\in \{Req, In, Out\}$
    Prio $\in \{\bot, 0, \ldots, \Delta_p - 1\}$
    $R$, Ok: Boolean; $PT \in [0 \ldots \ell + 1]$; $PPr \in [0 \ldots 2]$
repeat forever
    for all $q \in [0 \ldots \Delta_p - 1]$ do
        if (receive⟨ResT⟩ from $q$) then
            if (State $= Req$) $\wedge$ (|RSet| $<$ Need) then
                RSet $\leftarrow$ RSet $\cup \{q\}$
            else
                send⟨ResT⟩ to $q + 1$
            end if
        end if
        if (receive⟨PushT⟩ from $q$) then
            if (Prio $\neq \bot$) $\wedge$ (State $\neq Req \vee$ |RSet| $<$ Need)$\wedge$
              (State $\neq In$) then
                for all $i \in$ RSet do
                    send⟨ResT⟩ to $i + 1$
                end for
                RSet $\leftarrow \emptyset$
            end if
            send⟨PushT⟩ to $q + 1$
        end if
        if (receive⟨PrioT⟩ from $q$) then
            if Prio $= \bot$ then
                Prio $\leftarrow q$
            else
                send⟨PrioT⟩ to $q + 1$
            end if
        end if
        if (receive⟨ctrl, $C$, $R$, $PT$, $PPr$⟩ from $q$) then
            Ok $\leftarrow false$
            if ($q =$ Succ) $\wedge$ (myC $= C$) $\wedge$ (Succ $\neq 0$) then
                Succ $\leftarrow$ Succ $+ 1$
                Ok $\leftarrow true$
                if $R$ then
                    RSet $\leftarrow \emptyset$
                    Prio $\leftarrow \bot$
                end if
            end if
            if ($q = 0$) then
                Ok $\leftarrow true$
                if myC $\neq C$ then
                    Succ $\leftarrow \min(1, \Delta_p - 1)$
                    if $R$ then
                        RSet $\leftarrow \emptyset$
                        Prio $\leftarrow \bot$
                    end if
                end if
                myC $\leftarrow C$
            end if
            if Ok then
                $PT \leftarrow \min(PT + |\mathtt{RSet}|_q, \ell + 1)$
                if Prio $= q$ then
                    $PPr \leftarrow \min(PPr + 1, 2)$
                end if
                send⟨ctrl, myC, $R$, $PT$, $PPr$⟩ to Succ
            end if
        end if
    end for
    if (State $= Req$) $\wedge$ (|RSet| $\geq$ Need) then
        State $\leftarrow In$
        EnterCS()
    end if
    if (State $= In$) $\wedge$ ReleaseCS() then
        for all $i \in$ RSet do
            send⟨ResT⟩ to $i + 1$
        end for
        RSet $\leftarrow \emptyset$
        State $\leftarrow Out$
    end if
    if (Prio $\neq \bot$) $\wedge$ (State $\neq Req \vee$ |RSet| $\geq$ Need) then
        send⟨PrioT⟩ to Prio $+ 1$
        Prio $\leftarrow \bot$
    end if
end repeat
```

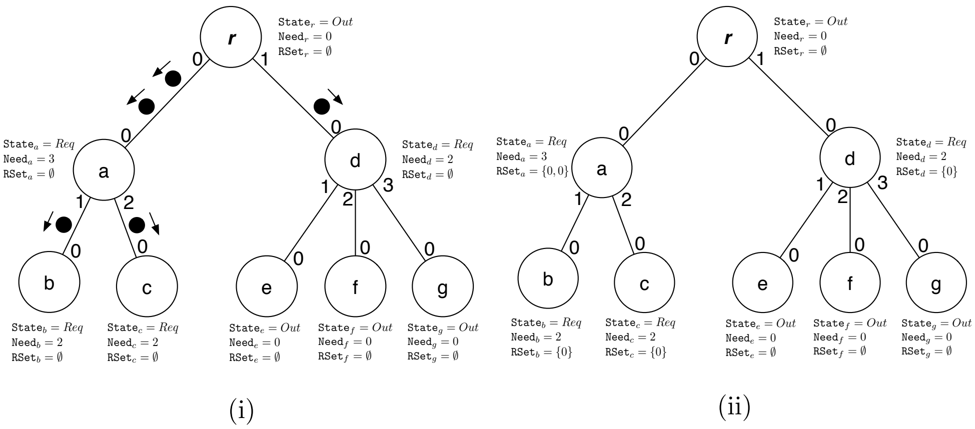
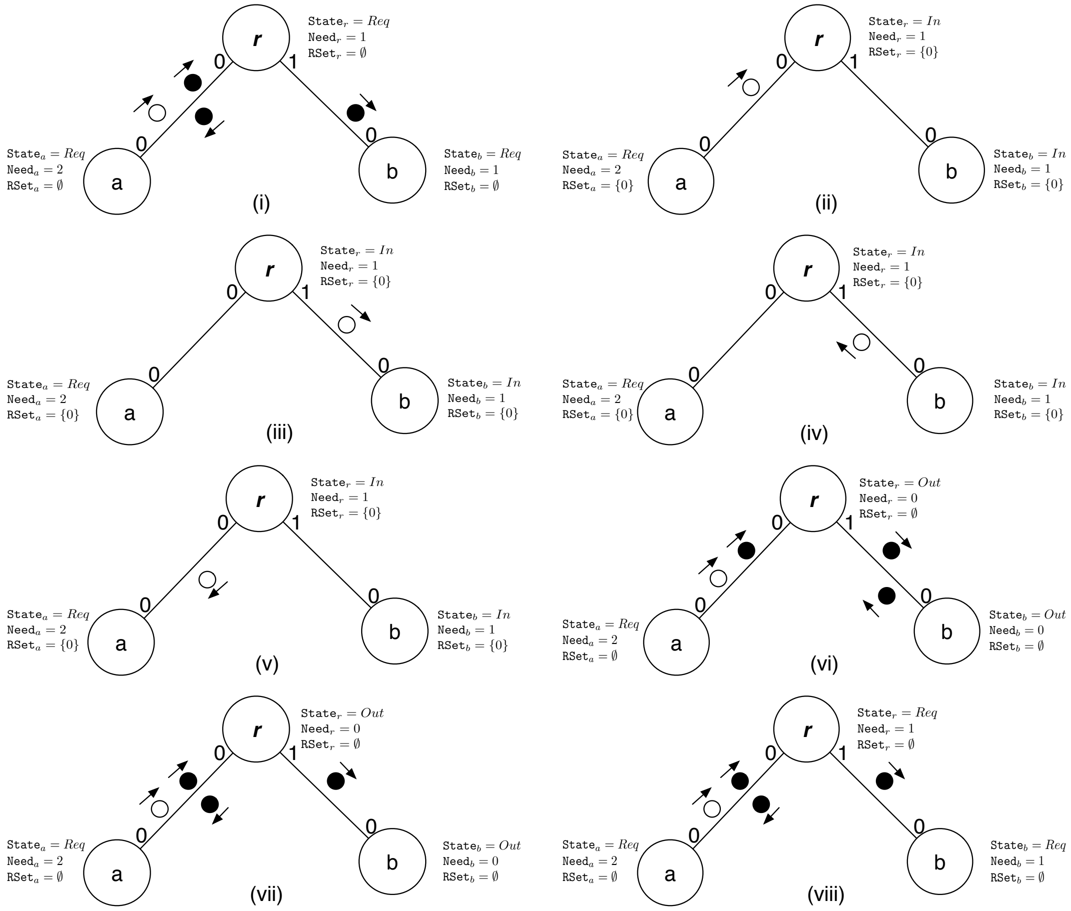


Fig. 2. Possible deadlock.

is not in the critical section from holding resource tokens forever. When a process receives the pusher, it releases all its reserved tokens, unless it is either in its critical section ($\mathtt{State} = \mathit{In}$) or is enabled to enter its critical section ($\mathtt{State} = \mathit{Req}$ and $|\mathtt{RSet}| \geq \mathtt{Need}$). In either case, it retransmits the pusher.



r
a
b
0
1
0
0
$\mathtt{State}_r = \mathit{Req}$
$\mathtt{Need}_r = 1$
$\mathtt{RSet}_r = \emptyset$
$\mathtt{State}_a = \mathit{Req}$
$\mathtt{Need}_a = 2$
$\mathtt{RSet}_a = \emptyset$
$\mathtt{State}_b = \mathit{Req}$
$\mathtt{Need}_b = 1$
$\mathtt{RSet}_b = \emptyset$
(i)
$\mathtt{State}_r = \mathit{In}$
$\mathtt{Need}_r = 1$
$\mathtt{RSet}_r = \{0\}$
$\mathtt{State}_a = \mathit{Req}$
$\mathtt{Need}_a = 2$
$\mathtt{RSet}_a = \{0\}$
$\mathtt{State}_b = \mathit{In}$
$\mathtt{Need}_b = 1$
$\mathtt{RSet}_b = \{0\}$
(ii)
$\mathtt{State}_r = \mathit{In}$
$\mathtt{Need}_r = 1$
$\mathtt{RSet}_r = \{0\}$
$\mathtt{State}_a = \mathit{Req}$
$\mathtt{Need}_a = 2$
$\mathtt{RSet}_a = \{0\}$
$\mathtt{State}_b = \mathit{In}$
$\mathtt{Need}_b = 1$
$\mathtt{RSet}_b = \{0\}$
(iii)
$\mathtt{State}_r = \mathit{In}$
$\mathtt{Need}_r = 1$
$\mathtt{RSet}_r = \{0\}$
$\mathtt{State}_a = \mathit{Req}$
$\mathtt{Need}_a = 2$
$\mathtt{RSet}_a = \{0\}$
$\mathtt{State}_b = \mathit{In}$
$\mathtt{Need}_b = 1$
$\mathtt{RSet}_b = \{0\}$
(iv)
$\mathtt{State}_r = \mathit{In}$
$\mathtt{Need}_r = 1$
$\mathtt{RSet}_r = \{0\}$
$\mathtt{State}_a = \mathit{Req}$
$\mathtt{Need}_a = 2$
$\mathtt{RSet}_a = \{0\}$
$\mathtt{State}_b = \mathit{In}$
$\mathtt{Need}_b = 1$
$\mathtt{RSet}_b = \{0\}$
(v)
$\mathtt{State}_r = \mathit{Out}$
$\mathtt{Need}_r = 0$
$\mathtt{RSet}_r = \emptyset$
$\mathtt{State}_a = \mathit{Req}$
$\mathtt{Need}_a = 2$
$\mathtt{RSet}_a = \emptyset$
$\mathtt{State}_b = \mathit{Out}$
$\mathtt{Need}_b = 0$
$\mathtt{RSet}_b = \emptyset$
(vi)
$\mathtt{State}_r = \mathit{Out}$
$\mathtt{Need}_r = 0$
$\mathtt{RSet}_r = \emptyset$
$\mathtt{State}_a = \mathit{Req}$
$\mathtt{Need}_a = 2$
$\mathtt{RSet}_a = \emptyset$
$\mathtt{State}_b = \mathit{Out}$
$\mathtt{Need}_b = 0$
$\mathtt{RSet}_b = \emptyset$
(vii)
$\mathtt{State}_r = \mathit{Req}$
$\mathtt{Need}_r = 1$
$\mathtt{RSet}_r = \emptyset$
$\mathtt{State}_a = \mathit{Req}$
$\mathtt{Need}_a = 2$
$\mathtt{RSet}_a = \emptyset$
$\mathtt{State}_b = \mathit{Req}$
$\mathtt{Need}_b = 1$
$\mathtt{RSet}_b = \emptyset$
(viii)

Fig. 3. Possible livelock.

The pusher protects the system from deadlock. However, it can cause *livelock*; an example is shown in Figure 3, for 2-out-of-3 exclusion in a tree of three processes. In Configuration (i), every process is a requester: r and b request one resource token and a requests two resource tokens. Also, every process has a resource token in one of its incoming channels, and none holds any resource token. Finally, the pusher is in the channel from a to r behind a resource token. Every process will collect the incoming resource token, and the system will reach the Configuration (ii) where r

and b execute their critical section while a is still waiting for a resource token and the pusher is reaching r. When r receives the pusher, it retransmits it to b, while keeping its resource token, as shown in Configuration (iii). Similarly, b receives the pusher while executing its critical section, and retransmits it immediately to r, as shown in Configuration (iv), after which r retransmits the pusher to a (Configuration (v)). Assume now that a receives the pusher while r and b leave their critical sections. We obtain Configuration (vi): a must release its resource tokens because of the pusher. In Configuration (vii), r directly retransmits the resource token it receives because it is not a requester. Finally, r and b again become requesters for one resource token in Configuration (viii), which is identical to Configuration (i). We can repeat this cycle indefinitely, and process a never satisfies its request.

To solve this problem, we add a *priority token* (message `PrioT`) whose goal is to overrule the pusher. If the system is in a legitimate state, there is exactly one priority token. A process which receives the priority token retransmits it immediately, unless it has an unsatisfied request. In this case, the process holds the priority token (the variable `Prio` is set from $\perp$ to the channel number from which the process receives the priority token) until its request is satisfied: the priority token will be released when the process enters its critical section. A process that holds the priority token does not release its reserved resource tokens when it receives the pusher: it only retransmits the pusher. As we will show later, this guarantees that the process will eventually satisfy its request.

Using these three types of tokens, we obtain a simple non-self-stabilizing `k`-out-of-ℓ exclusion protocol. To make it self-stabilizing, we need additional structure.

3.2. *A Controller for Self-Stabilization*

To achieve self-stabilization, we introduce one more type of token, the *controller*.

After a finite period of transient faults, some tokens may have disappeared or may been duplicated. To restore correct behavior, we need an additional self-stabilizing mechanism that regulates the number of tokens in the network: to achieve that, we use a mechanism similar to that introduced in [7] for self-stabilizing ℓ-exclusion on a ring. This mechanism is based on a snapshot/reset technique.

The controller is a special token (message `ctrl`) that counts the other tokens; when it returns to the root after one full circulation, the root learns the number of tokens of each type (resource, pusher, priority), and then adjusts these numbers as necessary.

The controller can also be effected by transient faults. We use Varghese's *counter flushing* [12] method to enforce *depth first token circulation* (DFTC) in the tree.

We now explain how the resource tokens are counted by the controller. (It counts the other types of tokens similarly.) We split the count of the resource tokens into two subcounts:

- *The "passed" tokens.* When a process holds some resource tokens that came

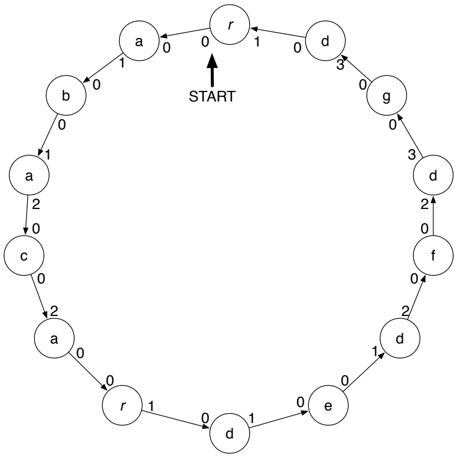


Fig. 4. Virtual Ring.

from channel i and receives the controller from the channel i, it retransmits the controller through channel $i + 1$ while keeping the resource tokens: in this case, we say that the controller *passes* these tokens in the virtual ring. Indeed, these tokens were ahead of the controller (in the virtual ring) before the process received the controller, and are behind afterward. The field PT of the controller message is used to compute the number of the passed resource tokens.

- *The tokens that are never passed by the controller.* These tokens are counted in the variable `SToken` maintained at the root. At the beginning of any circulation of the controller, the variable `SToken` is reset to 0. Until the end of the circulation of the controller, each time a resource token starts a new circulation (*i.e.*, the token leaves the root from channel 0), `SToken` is incremented.

When the controller terminates its circulation, the number of resource tokens in the network is equal to $PT +$ `SToken`, and the numbers of pusher tokens and priority tokens is likewise known to the root. Three cases are then possible:

- *The number of tokens is correct,* that is, there are ℓ resource tokens, one pusher token, and one priority token. In this case, the system is stabilized.
- *There are too few tokens.* In this case, the root creates the number of additional tokens needed at the end of the traversal; the system is then stabilized.
- *There are too many tokens of some type.* In this case, we reset the network. We mark the controller token with a special flag (the field R in the message `ctrl`). The root transmits the marked controller, erases its reserved tokens

as well as all the tokens it receives until the termination of the controller's traversal. Upon receiving the controller, every other process erases its reserved tokens. When the controller finishes its traversal, there is no token in the network. The root creates exactly ℓ resource tokens, one pusher token and one priority token; and we are done.

3.3. *Self-Stabilizing DFTC*

Using the counter flushing technique, we design a self-stabilizing $DFTC$ to implement the controller. The principle of counter flushing is the following: after transient faults, the token message can be lost; hence, the root must use a timeout mechanism to retransmit the token in case of deadlock. The timeout is managed using the function `RestartTimer`() (that allows it to reinitialize the timeout) and the predicate `TimeOut`() (which holds when a specified time interval is exceeded).[e]

Due to the use of the timeout, we must now deal with duplicated messages. Furthermore, arbitrary messages may exist in the network after faults (however the number of those message is assumed to be bounded). To distinguish the duplicates from the valid controller and to flush the system of corrupted messages, every process maintains a counter variable `myC` that takes values in $\{0 \ldots (n-1)(\mathtt{C_{MAX}}+1)\}$, and marks each message with that value. Every process also maintains a pointer `Succ` to indicate to which process it must send the token. The effects of the reception of a token message differ for the root and the other processes:

- The root considers a token message as valid when the message comes from `Succ` and is marked with a value c such that $\mathtt{myC} = c$. Otherwise, it simply ignores the message, meaning it does not retransmit it. If it receives a valid message, the root increments `Succ` (modulo Δ_r) and retransmits the token with the flag value `myC` to `Succ`, so that the valid token follows DFS order. If $\mathtt{Succ} = 0$, this means that the token just finished its previous circulation. As a consequence, the root increments `myC` (modulo $(n-1)\mathtt{C_{MAX}} + n$) before retransmitting the token.
- A non-root process p considers a message as valid in two cases: (1) When it receives a token message from its parent (channel 0) marked with a value c such that $\mathtt{myC} \neq c$ or (2) when it receives a token message from `Succ` and the message is marked with a value c such that $\mathtt{myC} = c$. In case (1), p sets `myC` to c and `Succ` to $\min(1, \Delta_p - 1)$ (*n.b.*, in case of a leaf process `Succ` is set to 0) before retransmitting the token message marked with `myC` to `Succ`. In case (2), p increments `Succ` (modulo Δ_p) and then sends the token marked with `myC` to `Succ` so that the valid token follows DFS order. In all other cases, p considers the message to be invalid. In the case of an invalid message coming from channel 0 with $\mathtt{myC} = c$, p does not consider

[e] We assume that this time interval is sufficiently large to prevent congestion.

the message in the computation, but retransmits it to prevent deadlock. In all other cases, p simply ignores the message.

Using this method, the root increments its counter `myC` infinitely often and, due to the size of `myC`'s domain, the `myC` variable of the root eventually takes a value that does not exist anywhere else in the system (because the number of possible values initially in the system is bounded by $(n-1)\mathtt{C_{MAX}}+n$). In this case, the token marked with the new value will be considered as valid by every process. Until the end of that traversal, the root will ignore all other token messages. At the end of the traversal, the system will be stabilized.

3.4. *Dealing with Bounded Memory*

Due to the use of reset, the root does not need to know the exact number of tokens at the end of the controller's traversals. Actually, the root must only know if the number of tokens is too high, or the number of tokens it needs to add if the number is too low. Hence, the counting variables can be bounded by $\ell+1$ for the resource tokens and by 2 for the other types of token. The fact that a variable is assigned to its maximum value will mean that there are too many tokens in the network and so a reset must be started. Otherwise, the value of the counting variable will state whether there is a deficient number of tokens, and in that case, how many must be added. For any assignment to one of these bounded variables, the value is set to the minimum between its new computed value and the maximum value of its domain.

4. Correctness

In this section, we prove that our protocol is a self-stabilizing `k`-out-of-ℓ exclusion protocol. The proof is done in three steps: (1) Recall that the controller part of our protocol is a self-stabilizing DFS token circulation. (2) We show that once the controller is stabilized to DFS token circulation, the system eventually stabilizes to the expected number of different tokens. (3) We show that once the system contains the expected number of tokens, the system stabilizes to the `k`-out-of-ℓ exclusion specification.

In our protocol, when a process receives a `ctrl` message, either it considers the message as valid or not. The process takes account of the messages for computations only when they are valid. Assume that a process p receives a `ctrl` message marked with the flag value c from channel q. Process p considers this message as valid if and only if $(q=\mathtt{Succ}_p \wedge c=\mathtt{myC}_p) \vee (p\neq r \wedge (q=0 \wedge c\neq \mathtt{myC}_p))$.

In the following, we call any `ctrl` message a *control token*. Each time a process receives a valid `ctrl` message, it makes some local computations, and then sends another `ctrl` message. In the case of a non-root process, the sent message is marked with the same flag as the received message: we consider it to be the same control token. In the case of the root, the sent message is marked either with the same value or with a new one. In the former case, we consider it to be the same control token, while in the latter case, we consider the received control token to have terminated its traversal, and the transmitted control token to be new.

To implement the control part, we use the counter flushing techniques introduced by Varghese in [12]. Hence, from [12], we can deduce the following lemma:

Lemma 2. *Starting from any configuration, the system converges to a configuration at which:*

1. *There exists at most one valid control token in the network.*
2. *The root regularly creates a new valid control token.*
3. *Any valid control token visits all processes in DFS order.*

Remark 3. *In our protocol, only the valid control tokens are considered in the computations. Hence, from now on, we only consider the valid control tokens and we simply refer to them as* control tokens.

We now show that starting from any configuration, the system eventually contains the expected number of each type of token.

Note that each resource token is either in a link (in this case, the token is said to be *free*) or it is stored in the `RSet` of a process (in this case, the token is said to be *reserved*). Hence, at any time the number of resource tokens in the network is equal to the sum of the sizes of the `RSet` multisets plus the number of free resource tokens. Similarly, at any time, the number of priority tokens is equal to the number of processes satisfying `Prio` $\neq \perp$ plus the number of free priority tokens. Finally, as a process cannot store any pusher token, the number of pusher tokens is equal to the number of free pusher tokens.

Lemma 4. *Let γ be the first configuration after the control part is stabilized. If, after γ, the root creates a control token whose reset field R is true, then the system contains no resource, priority, and pusher token at the end of the traversal of the control token.*

Proof. Consider any control token created by the root after configuration γ. Assume that the reset field R of the control token is set to true. Then, the `Reset` variable of the root is also true (see Line 74 in Algorithm 1). $\texttt{Reset}_r$ remains true until the control token terminates its traversal. Hence, during the traversal, any token (except the control token) that is received by the root is ignored by the root and so disappears from the network (see Lines 11, 21, and 36 in Algorithm 1). Also, during its traversal, each process erases all tokens (except the control token) it holds when visited by the control token (see Line 49 in Algorithm 1, and Lines 38, and 47 in Algorithm 2). Hence, every resource, priority, or pusher token is either erased at a process when the process is visited by the control token, or is pushed to the root and then disappears. At the end of the traversal of the control token, the system contains no resource, priority, or pusher tokens. □

Lemma 5. *Let γ be the first configuration after the control part is stabilized. When a control token created by the root after γ terminates its traversal, we have:*

- *If $PT + \texttt{SToken}_{\rm r} > \ell$, then there are more than ℓ resource tokens in the network.*
- *If $PT + \texttt{SToken}_{\rm r} \leq \ell$, then there are exactly $PT + \texttt{SToken}$ resource tokens in the network.*

Proof. Consider any control token created by the root after configuration γ. There are two cases:

- *The reset field R of the control token is true.* By Lemma 4, there is no resource token in the network when the control token terminates its circulation. So, to prove the lemma in this case, we must show that $PT + \texttt{SToken}_r = 0$ at the end of the circulation.

  First, $\texttt{SToken}_r$ is reset to 0 (Line 64) before the control token starts its circulation (Line 74). Also, $\texttt{Reset}_r$ is true when the control token starts its circulation (see Line 74 in Algorithm 1). Thus, until termination of the circulation, r ignores any resource tokens it receives (see Lines 11, 21, and 36) and so $\texttt{SToken}_r$ is still equal to 0 at the end of the control token circulation.

  Consider now the PT field of the control token. Before the start of the control token circulation, r executes the following action: RSet is set to $\emptyset$ (Line 50 in Algorithm 1), PT is set to 0 (Line 67 in Algorithm 1), and, as a consequence, PT is set to $\min(0, \ell + 1)$ (Line 70 in Algorithm 1). So, at the start of the control token circulation, the control token is sent with its PT field equal to 0. Since the reset field R of the control token is true, each time the control token arrives at a process, the process resets its RSet variable to $\emptyset$ (see Lines 50 in Algorithm 1, Lines 39, and 48 in Algorithm 2) before setting PT to $\min(PT + |\texttt{RSet}|_q, \ell + 1)$ (see Line 70 in Algorithm 1 and Line 55 in Algorithm 2) and then retransmitting the token. Hence, PT remains equal to 0 until the end of the circulation.

  When the control token terminates its circulation, $PT + \texttt{SToken}_r = 0$, and we are done.
- *The reset field R of the control token is false.* In this case, we can remark that no resource token is erased during the circulation of the control token, because both R and $\texttt{Reset}_r$ are false.

  (*) We now show that any resource token is counted at most once during the circulation of the control token. Due to the FIFO quality of the links and the fact that when the control token is received by a process, the process receives no other message before retransmitting the control token, we have the following property: a resource token is passed by the control token at most once during a circulation. So, during the circulation, either the resource token is counted into the PT field of the control token when the resource token is passed by the control token (see Line 70 in Algorithm 1 and Line 55 in Algorithm 2) or it is counted at the root when it terminates a loop of the virtual ring (Line 16). Hence, any resource token is counted at most once.

(**) Finally we show, by contradiction, that any resource token is counted at least once during the circulation of the control token. Assume that a resource token is not counted during that circulation. Then, the resource token is never passed by the control token. The links are FIFO, and when the control token is received by a process, the process receives no other message before retransmitting the control token. Therefore, the resource token is always ahead of the control token in the virtual ring. As a consequence, the resource token is eventually counted at the root when it terminates a loop of the virtual ring (Line 16), contradiction.

From (*), we know that if $PT + \texttt{SToken}_r > \ell$ at the end of the control token circulation, then there are more that ℓ resource tokens in the network. From (*) and (**), we know that if $PT + \texttt{SToken}_r \leq \ell$ at the end of the control token circulation, then there are exactly $PT + \texttt{SToken}_r$ resource tokens in the network, and we are done. □

Following similar reasoning, we obtain the following two lemmas:

Lemma 6. *Let γ be the first configuration after the control part is stabilized. When a control token created by the root after γ terminates its traversal, we have:*

- *If $\texttt{SPrio}_\mathrm{r} + PPr > 1$, there is more that one priority token in the network.*
- *If $\texttt{SPrio}_\mathrm{r} + PPr \leq 1$, there are exactly $\texttt{SPrio} + PPr$ priority tokens in the network.*

Lemma 7. *Let γ be the first configuration after the control part is stabilized. When a control token created by the root after γ terminates its traversal, we have:*

- *If $\texttt{SPush}_\mathrm{r} > 1$, then there is more that one pusher token in the network.*
- *If $\texttt{SPush}_\mathrm{r} \leq 1$, then there are exactly $\texttt{SPush}_\mathrm{r}$ pusher tokens in the network.*

Lemma 8. *Starting from any configuration, the system eventually reaches a configuration from which there always exist exactly ℓ resources tokens.*

Proof. Let γ be the first configuration after the control part is stabilized. Consider any control token created by the root after γ. There are two cases:

- *$PT + \texttt{SToken}_\mathrm{r} \leq \ell$ at the end of the control token traversal.* Then, $\texttt{Reset}_r$ is set to false. (Line 48 in Algorithm 1) and, as a consequence, the reset field of the next control token will be false (Line 74 of Algorithm 1). Hence, no resource token will be erased during the next circulation of a control token. If $PT + \texttt{SToken}_r < \ell$, then exactly $\ell - (PT + \texttt{SToken}_r)$ are created (see Lines 56 to 59 in Algorithm 1). Hence, the number of resource tokens will be exactly equal to ℓ at the beginning of the next control token circulation. By Lemma 5, $PT + \texttt{SToken}_r$ will be equal to ℓ at the end of the next control token circulation, no resource token will be added. Any later circulation of the control token cannot change the number of resource tokens. Hence, the system will contain ℓ resource tokens forever.

- $PT + \texttt{SToken}_\mathrm{r} > \ell$ *at the end of the control token traversal.* Then, $\texttt{Reset}_r$ is set to true (Line 48 in Algorithm 1) and, as a consequence, the reset field of the next control token will be true (Line 74 of Algorithm 1). By Lemmas 4 and 5, we can reduce to the previous case when the circulation of the next control token terminates, and we are done. □

Following similar reasoning, we can deduce from Lemmas 4, 6, and 7, the following two lemmas:

Lemma 9. *Starting from any configuration, the system eventually reaches a configuration from which there always exists one priority token.*

Lemma 10. *Starting from any configuration, the system eventually reaches a configuration after which there always exists one pusher token.*

We now show that once the system contains the correct number of each type of token, the system stabilizes to the k-out-of-ℓ exclusion specification.

Lemma 11. *Starting from any configuration, every process receives a pusher token infinitely many times.*

Proof. By Lemmas 8, 9, and 10, starting from any configuration, the system eventually reaches a configuration γ after which there are always exactly ℓ resource tokens, one priority token, and one pusher token in the network. From γ, the system is then never again reset. So, from γ, the unique pusher token of the system always follows DFS order. Each time a process receives the pusher token, it retransmits it in finite time. Hence, every processes receives it infinitely often and the lemma holds. □

Lemma 12. *Starting from any configuration, every process receives a priority token infinitely many times.*

Proof. By Lemmas 8, 9, and 10, starting from any configuration, the system eventually reaches a configuration γ from which there are ℓ resource tokens, one priority token, and one pusher token in the network. From γ, the system is never again reset. So from γ, the unique priority token of the system always follow the DFS order.

By way of contradiction, assume that, from γ, a process eventually stops receiving the priority token. Since the priority token circulates in DFS order and traverses each link in finite time, we can deduce that some other process p eventually holds it forever. In this case, p is a requester and its request is never satisfied. Now, by

Lemma 11 every other process receives the pusher token infinitely often. Thus, each other process retransmits the resource tokens it holds within finite time, because it eventually either satisfies its request, executes its critical sections, and then releases its tokens, or does not satisfy its request, but receives the pusher, and then releases its resource tokens. Similarly the resource tokens always follows DFS order after γ. Hence, p receives resource tokens infinitely many times, and never releases the the priority token even if it receives the pusher token. Since $\mathtt{k} \leq \ell$, the request of p is eventually satisfied, contradiction. □

Lemma 13. *Starting from any configuration, every process receives resource tokens infinitely many times.*

Proof. By Lemmas 8, 9, and 10, starting from any configuration, the system eventually reaches a configuration γ from which there are ℓ resource tokens, one priority token, and one pusher token in the network. After γ, the system is then never again reset. Thus, after γ, the resource tokens of the system always follow DFS order.

Assume, by way of contradiction, that some process only receives resource tokens finitely many times. This implies that every resource token is eventually held forever by some process. Consider one process that holds at least one resource token forever. By Lemma 11, that process cannot hold the priority token forever. When it releases the priority token, either its request is satisfied, it executes the critical section within finite time, and then releases its resource tokens, or it is not a requester and thus must release its resource token. Either case is a contradiction, and we are done. □

Lemma 14. *Starting from any configuration, the fairness property of the* $\mathtt{k}$*-out-of-*ℓ *exclusion specification is eventually satisfied.*

Proof. Assume that there a request by some process p that is never satisfied. By Lemmas 8, 9, and 10, starting from any configuration, the system eventually reaches a configuration γ from which there are always ℓ resource tokens, one priority token, and one pusher token into the network. After γ, the system is then never again reset. Hence, after γ, if p holds the priority token, it releases it only if its request is satisfied. By Lemma 13, p eventually receives the priority token. Again by Lemma 13, p eventually releases the priority token, and so its request must have been satisfied, contradiction. □

Lemma 15. *Starting from any configuration, the safety property of the* $\mathtt{k}$*-out-of-*ℓ *exclusion specification is eventually satisfied.*

Proof. By Lemma 8, there are eventually exactly ℓ resource tokens in the network. Hence, eventually, exactly ℓ resource unit are available in the system.

Finally, any process p that initially holds some resource tokens eventually releases them because either is is not a requester or it eventually satisfies its request by Lemma 14. Hence, eventually p sets $\texttt{RSet}$ to $\emptyset$ and then $|\texttt{RSet}| \leq \texttt{Need}$ forever because each time p receives a resource token while $|\texttt{RSet}| \geq \texttt{Need}$, it directly retransmits it (see Lines 11 to 20 in Algorithm 1 and Lines 9 to 16 in Algorithm 2). Now, $\texttt{Need}$ is always less than or equal to $\texttt{k}$. Hence, every process eventually only uses at most $\texttt{k}$ resource tokens (units) simultaneously. □

Lemma 16. *Starting from any configuration, the efficiency property of the* k*-out-of-*ℓ *exclusion specification is eventually satisfied.*

Proof. We use the definition of efficiency given in [2]. We prove that starting from any configuration, $(\texttt{k}, \ell)$*-liveness* is eventually satisfied.

Consider the configuration γ after which: (1) there are always ℓ resource tokens, one priority token, and one pusher token; and (2) the safety properties of the k-out-of-ℓ exclusion are satisfied (such a configuration exists by Lemmas 8, 9, 10, and 15).

Assume that after γ, the system reaches a configuration γ' after which there is a subset I of processes such that every process in I executes its critical section forever (in this case they hold some resource units forever). Let α be the total number of resource units held forever by the processes in I.

Assume then that there are some processes not in I that request some resource units and each of these processes requests at most $\ell - \alpha$ resource units.

The priority token follows DFS order. Since every process in I executes the critical section forever, none of these processes keeps the priority token forever (see Lines 93 in Algorithm 1 and 74 in Algorithm 2). Finally, every non-requester directly retransmits the priority token when it receives it (see Line 93 in Algorithm 1 and Line 74 in Algorithm 2). Hence, there is a requesting process p which is not in I that eventually receives the priority token. From that point, p will release it only after its request is satisfied (see Line 93 in Algorithm 1 and Line 74 in Algorithm 2). As a consequence, p will keep every resource token it receives, even if it receives the pusher token. Checking the proof of Lemma 11, we can see that Lemma 11 still holds even if some processes execute the critical section forever. So, by Lemma 11 every process that is not in $I \cup \{p\}$ receives the pusher token infinitely often, and so cannot hold resource tokens forever. Finally, every process in I directly retransmits the resource tokens it receives while it is executing the critical section because they satisfy $|\texttt{RSet}| \geq \texttt{Need}$ by Lemma 15 (see Lines 11 to 20 in Algorithm 1 and Lines 9 to 16 in Algorithm 2). So, p eventually receives the resource tokens it needs to perform the critical section (remember that p requests at most $\ell - \alpha$ resource units) and we are done. □

From Lemmas 15, 14, and 16, we obtain:

Theorem 17. *The protocol proposed in Algorithms 1 and 2 is a self-stabilizing* k*-out-of-*ℓ *exclusion protocol for tree networks.*

5. Complexity Analysis

We start the complexity analysis by computing the waiting time of our solution, a crucial parameter in resource allocation. We then study the stabilization time.

5.1. *Waiting Time*

Theorem 18. *Once the protocol proposed in Algorithms 1 and 2 is stabilized, the waiting time is $\ell \times (2n-3)^2$ in the worst case.*

Proof. We first show that the waiting time of a requesting process that holds the priority token is $\ell \times (2n-3)$ in the worst case. Consider a process p that requests some resource units and holds the priority token. In the worst case, p appears only once in the virtual ring defined by the DFS order (if p is a leaf). Also in the worst case, the ℓ resource tokens may traverse the entire virtual ring before p receives the tokens it needs. The virtual ring can contain up to $(2n-3)$ processes in addition to p. Any resource token may satisfy one request each time it traverses a process (in the worst case, each process other than p always requests one token). Hence, the ℓ resource tokens may satisfy up to $\ell \times (2n-3)$ requests before p satisfies its request.

Using similar reasoning, we can see that a requesting process could wait until the priority token traverses the whole virtual ring (up to $2(n-2)$ nodes) before it satisfy its request; during that time, up to $\ell \times (2n-3)^2$ requests can be satisfied, and we are done. □

5.2. *Stabilization time*

For computing the stabilization time, we consider a notion of *round* that is similar to the one used in [12]: we assume that every process executes an iteration of its "repeat forever" loop in *one round*. Also, any message stored in a link is delivered in *one round*.

In [12], authors define a *rotation time* to be the time for a token to traverse the network with a one-round delay at each node and each link. Here, the number of edges is the virtual ring is $2(n-1)$. So, in our system, a rotation time is equal to $4(n-1)$ rounds.

From the results in [12], we know that the controller part of our protocol stabilizes to a single token circulation in at most three rotation times. More precisely, after at most three rotation times, the root creates a controller token that is unique in the system.

So, after one more rotation time, the root knows the number of tokens of each type. In the worst case, the root then starts a reset of the network and after another rotation time, the system is stabilized.

Hence, the system stabilizes in at most 5 rotation times, and we have the following theorem:

Theorem 19. *The protocol proposed in Algorithms 1 and 2 stabilizes in at most*

20$(n-1)$ *rounds.*

6. Conclusion and Future Work

In this paper, we propose the first (deterministic) self-stabilizing distributed k-out-of-ℓ exclusion protocol for asynchronous oriented tree networks. The proposed protocol uses a realistic model, the message-passing model. The only restriction we make is to assume that the links initially contain at most a bounded number of arbitrary messages, where the bound is known. We make this assumption to obtain a solution that uses bounded memory per process (see the results in [6]). However, if we assume unbounded process memory, our solution can be easily adapted to work without assumptions on channels (following the method of [8]).

The main interest in dealing with an oriented tree is that solutions on the oriented tree can be directly mapped to solutions for arbitrary rooted networks by composing the protocol with spanning tree construction.

There are several possible extensions of our work. On the theoretical side, one can investigate whether the waiting time of our solution ($\ell \times (2n-3)^2$) or the stabilization time ($20(n-1)$ rounds) can be improved. Possible extension to networks where processes are subject to other failure patterns, such as process crashes, remains open. On the practical side, our solution is designed using a realistic model and can be extended to arbitrary rooted networks. Implementing our solution in a real network is a future challenge.

References

[1] A K Datta, R Hadid, and V Villain. A new self-stabilizing k-out-of-l exclusion algorithm on rings. In Shing-Tsaan Huang and Ted Herman, editors, *Self-Stabilizing Systems*, volume 2704 of *Lecture Notes in Computer Science*, pages 113–128. Springer, 2003.

[2] AK. Datta, R. Hadid, and V. Villain. A self-stabilizing token-based k-out-of-l exclusion algorithm. *Concurrency and Computation: Practice and Experience*, 15(11-12):1069–1091, 2003.

[3] EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.

[4] Shlomi Dolev. *Self-Stabilization.* The MIT Press, March 2000.

[5] M J Fischer, N A Lynch, J E Burns, and A Borodin. Distributed fifo allocation of identical resources using small shared space. *ACM Trans. Program. Lang. Syst.*, 11(1):90–114, 1989.

[6] Mohamed G. Gouda and Nicholas J. Multari. Stabilizing communication protocols. *IEEE Trans. Computers*, 40(4):448–458, 1991.

[7] Rachid Hadid and Vincent Villain. A new efficient tool for the design of self-stabilizing l-exclusion algorithms: The controller. In Ajoy Kumar Datta and Ted Herman, editors, *WSS*, volume 2194 of *Lecture Notes in Computer Science*, pages 136–151. Springer, 2001.

[8] Shmuel Katz and Kenneth J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7(1):17–26, 1993.

[9] Y Manabe, R Baldoni, M Raynal, and S Aoyagi. k-arbiter: A safe and general scheme for h-out of-k mutual exclusion. *Theor. Comput. Sci.*, 193(1-2):97–112, 1998.

[10] Y Manabe and N Tajima. (k)-arbiter for h-out of-k mutual exclusion problem. In *ICDCS*, pages 216–223, 1999.

[11] M Raynal. A distributed solution to the k-out of-m resources allocation problem. In F K H A Dehne, F Fiala, and W W Koczkodaj, editors, *ICCI*, volume 497 of *Lecture Notes in Computer Science*, pages 599–609. Springer, 1991.

[12] George Varghese. Self-stabilization by counter flushing. *SIAM J. Comput.*, 30(2):486–510, 2000.