Self-Stabilizing $\mathtt{k}$-out-of-ℓ Exclusion on Tree Networks

Ajoy K. Datta
School of Computer Science, University of Nevada

Stéphane Devismes
VERIMAG, Université Joseph Fourier

Florian Horn
LIAFA, Université Paris Denis Diderot

Lawrence L. Larmore
School of Computer Science, University of Nevada

Abstract

In this paper, we address the problem of $\mathtt{k}$-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 $\mathtt{k}$ units ($1 \leq \mathtt{k} \leq \ell$). We propose the first deterministic self-stabilizing distributed $\mathtt{k}$-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, self-stabilization, resource allocation, $\mathtt{k}$-out-of-ℓ exclusion, oriented tree networks.*

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 an agent affects their availability for the other users. In the aforementioned 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 [12]: the critical section can be executed by at most one process at any time. The ℓ-exclusion property [6] 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 [14] allows us to deal with such requests; requests may vary from 1 to $\mathtt{k}$ units of a given resource, where $1 \leq \mathtt{k} \leq \ell$.

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 [5] if, after transient faults hit the system and place it in some arbitrary global state, the systems recovers from this catastrophic situation 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, there is no prior protocol of this type 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), such as given in [1, 4]. In the other hand, the complexity of the solution comes from the fact that the problem is a generalization of mutual exclusion. This is exacerbated by the difficulty of obtaining self-stabilizing solutions in message-passing system (the more realistic model), as underlined by the impossibility result of Gouda and Multari [7].

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.

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 permissions from all processes [14], or from the processes constituting its quorum [10, 11]. There exist two self-stabilizing solutions for $\mathtt{k}$-out-of-ℓ exclusion on the oriented rooted ring [2, 3]. These solutions are based on circulation of ℓ tokens, where each token corresponds to a resource unit.

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. We conclude in Section 4.

Due to the lack of space, the technical proofs have been omitted. For further details, see the technical report online at `http://hal.archives-ouvertes.fr/hal-00344193/fr/`.

2. Preliminaries

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 and 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. [1] The local algorithm is structured as 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 $\{0 \ldots \Delta_p - 1\}$; by an abuse of notation, we may refer to a neighbor q of p by the label of p's channel to q. We assume that the channels are *reliable*, meaning that no message can be lost (after the end of the transient faults) and *FIFO*, meaning that messages are received in the order they are sent. We also assume that each channels initially contains some arbitrary messages, but not more than a given bound $\mathtt{C_{MAX}}$.[2]

A message is of the following form: $\langle \textit{type}, \textit{value}\rangle$. The *value* field is omitted if the message does not carry any value. A message may also contain more than one 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 $\gamma_0\gamma_1 \ldots \gamma_i \ldots$ 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.

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

[2] This assumption is required to obtain a deterministic self-stabilizing solution working with bounded process memory; see [7].

$\mathtt{k}$-out-of-ℓ 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:** As many requests as possible must be satisfied simultaneously.

The above mentioned notion of *efficiency* is difficult to define precisely. A convenient parameter was introduced in [3] to formally characterize efficiency: $(\mathtt{k}, \ell)$-*liveness*, defined as follows. 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.

Waiting Time. The *waiting time* [13] is the maximum number of times that all processes can enter in the critical section before some process p, starting from the moment p requests the critical section.

Interface. In any $\mathtt{k}$-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:

- $\mathtt{State} \in \{\textit{Req}, \textit{In}, \textit{Out}\}$. $\mathtt{State} = \textit{Req}$ means that the application is requesting some resource units. $\mathtt{State}$ switches from *Req* to *In* when the application is allowed to access to the requested resource units. $\mathtt{State}$ switches from *In* to *Out* when the requested resource units are released into the system. The switching of $\mathtt{State}$ from *Req* to *In* and from *In* to *Out* is managed by the $\mathtt{k}$-out-of-ℓ exclusion protocol itself; while the switching from *Out* to *In* is managed by the application. Other transitions (for instance, *In* to *Req*) are forbidden.
- $\mathtt{Need} \in \{0 \ldots \mathtt{k}\}$, the number of resource units currently being requested by the application.

- $\texttt{EnterCS}()$: *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).
- $\texttt{ReleaseCS}()$: *Boolean*. This predicate holds if and only if the application is not executing its critical section.

Self-Stabilization [5]. 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 $\gamma \in C_2$ and any execution whose initial configuration is γ, the execution contains a configuration of $\mathcal{C}_1$.

Definition 1 *A protocol $\mathcal{P}$ is* self-stabilizing *for the specification SP in a system $\mathcal{S}$ if there exists a non-empty subset of $\mathcal{L}$ such that:*

- *Any execution of $\mathcal{P}$ in $\mathcal{S}$ starting from a configuration of $\mathcal{L}$ satisfies SP (*Closure Property*).*
- *$\mathcal{L}$ is an attractor for $\mathcal{C}$ (*Convergence Property*).*

3. Protocol

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

A non-fault-tolerant protocol. The basic principle of our protocol is to use ℓ circulating *resource tokens* (the $\texttt{ResT}$ 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*.

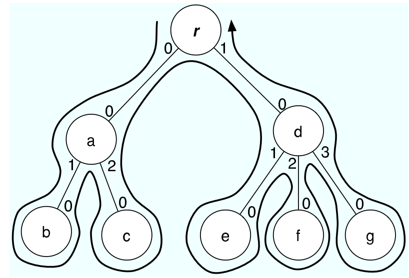

Figure 1. Depth-first token circulation on oriented trees.

As explained Section 2, the requests are managed by the variables $\texttt{State}$ and $\texttt{Need}$. Each process also uses the multiset[3] variable $\texttt{RSet}$ to collect the tokens; the collected tokens are said to be "reserved." While $\texttt{State} = \textit{Req}$ and $|\texttt{RSet}| < \texttt{Need}$, a process collects all tokens it receives; it also stores in $\texttt{RSet}$ 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 $\texttt{State} = \textit{Req}$ and $|\texttt{RSet}| \geq \texttt{Need}$, it enters the critical section: $\texttt{State}$ is set to *In* and the function $\texttt{EnterCS}()$ is called. Once the critical section is done (*i.e.*, when $\texttt{State} = \textit{In}$ and the predicate $\texttt{ReleaseCS}()$ holds) $\texttt{State}$ is set to *Out*, all tokens in $\texttt{RSet}$ are retransmitted, and $\texttt{RSet}$ is set to $\emptyset$. When a process receives a token it does not need, it immediately retransmits it.

Unfortunately, such a simple protocol does not always guarantee liveness. Figure 2 shows a case where liveness is not maintained. In this example, there are five resources tokens (*i.e.*, $\ell = 5$) and each process can request up to three tokens (*i.e.*, $\texttt{k} = 3$). In the configuration shown on the left side of the figure, processes a, b, c, and d request more tokens than they will receive. This configuration will lead to the deadlock configuration shown on the right side of the figure: 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 $\texttt{PushT}$). 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 is not in the critical section from holding resource tokens forever. When a process receives the pusher, it releases all its reserved tokens, unless if it is either in its critical section ($\texttt{State} = \textit{In}$) or is enabled to enter its critical section ($\texttt{State} = \textit{Req}$ and $|\texttt{RSet}| \geq \texttt{Need}$). In either case, it retransmits the pusher.

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

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

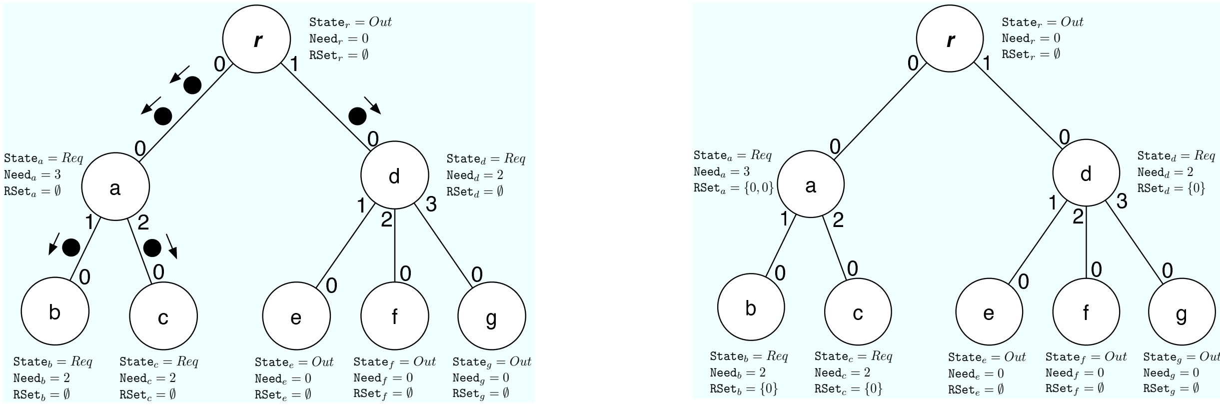

Figure 2. Possible deadlock.

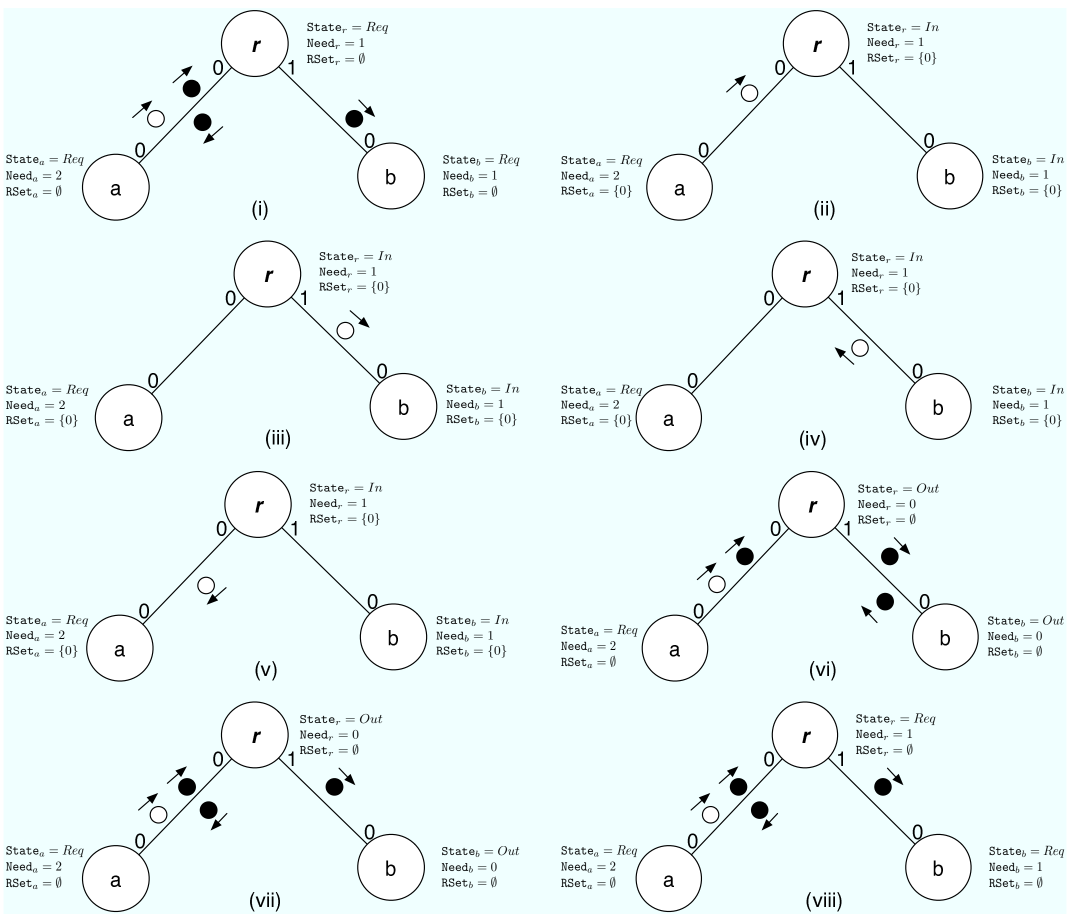

Figure 3. Possible livelock.

Algorithm 1 $\mathtt{k}$-out-of-ℓ exclusion on oriented trees, local algorithm for the root *r*

```
variables:
    C, myC ∈ [0...2(n−1)(C_MAX + 1)]; Succ ∈ [0...Δr − 1]
    RSet: multiset of at most k values taken in [0...Δr − 1]
    Need∈[0...k]; State∈{Req,In,Out}; Prio∈{⊥, 0, ..., Δr − 1}
    R, Reset: Booleans; SToken, PT∈[0...ℓ + 1]
    SPush, SPrio, PPr ∈ [0...2]
repeat forever
    for all q ∈ [0...Δr − 1] do
        if (receive⟨ResT⟩ from q) ∧¬Reset then
            if (State = Req) ∧ (|RSet| < Need) then
                RSet ← RSet ∪ {q}
            else
                if q = Δr − 1 then
                    SToken ← min(SToken + 1, ℓ + 1)
                end if
                send⟨ResT⟩ to q + 1
            end if
        end if
        if (receive⟨PushT⟩ from q) ∧¬Reset then
            if (Prio ≠⊥) ∧ (State ≠ Req ∨ |RSet| < Need)∧
             (State ≠ In) then
                for all i ∈ RSet do
                    if i = Δr − 1 then
                        SToken ← min(SToken + 1, ℓ + 1)
                    end if
                    send⟨ResT⟩ to i + 1
                end for
                RSet ← ∅
            end if
            if q = Δr − 1 then
                SPush ← min(SPush + 1, 2)
            end if
            send⟨PushT⟩ to q + 1
        end if
        if (receive⟨PrioT⟩ from q) ∧¬Reset then
            if Prio =⊥ then
                Prio ← q
            else
                send⟨PrioT⟩ to q + 1
            end if
        end if
        if (receive⟨ctrl, C, R, PT, PPr⟩ from q) then
            if (q = Succ) ∧ (myC = C) then
                Succ ← Succ + 1
                if Succ = 0 then
                    myC ← myC + 1
                    Reset ← (PT + SToken > ℓ)∨
                      (PPr + SPrio > 1) ∨ (SPush > 1)
                    if Reset then
                        RSet ← ∅
                        Prio ←⊥
                    else
                        if PPr + SPrio < 1 then
                            send⟨PrioT⟩ to 0
                        end if
                        while PT + SToken < ℓ do
                            send⟨ResT⟩ to 0
                            SToken ← min(SToken + 1, ℓ + 1)
                        end while
                        if SPush < 1 then
                            send⟨PushT⟩ to 0
                        end if
                    end if
                    SToken ← 0
                    SPrio ← 0
                    SPush ← 0
                    PT ← 0
                    PPr ← 0
                end if
                PT ← min(PT + |RSet|_q, ℓ + 1)
                if Prio = q then
                    PPr ← min(PPr + 1, 2)
                end if
                send⟨ctrl, myC, Reset, PT, PPr⟩ to Succ
                RestartTimer()
            end if
        end if
    end for
    if (State = Req) ∧ (|RSet| ≥ Need) then
        State ← In
        EnterCS()
    end if
    if (State = In) ∧ ReleaseCS() then
        for all i ∈ RSet do
            if i = Δr − 1 then
                SToken ← min(SToken + 1, ℓ + 1)
            end if
            send⟨ResT⟩ to i + 1
        end for
        RSet ← ∅
        State ← Out
    end if
    if (Prio ≠⊥) ∧ (State ≠ Req ∨ |RSet| ≥ Need) then
        if Prio = Δr − 1 then
            SPrio ← min(SPrio + 1, 2)
        end if
        send⟨PrioT⟩ to Prio + 1
        Prio ←⊥
    end if
    if TimeOut() then
        send⟨ctrl, myC, Reset, 0, 0⟩ to Succ
        RestartTimer()
    end if
end repeat
```

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 cancel the effect of 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

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

```
variables:
    C, myC ∈ [0 ... 2(n − 1)(C_MAX + 1)]; Succ ∈ [0 ... Δ_p − 1]
    RSet: multiset of at most k values taken in [0 ... Δ_p − 1]
    Need∈[0 ... k]; State∈{Req,In,Out}; Prio∈{⊥, 0, ... , Δ_p − 1}
    R, Ok: Booleans; PT∈[0 ... ℓ + 1]; PPr∈[0 ... 2]
repeat forever
  for all q ∈ [0 ... Δ_p − 1] do
    if (receive⟨ResT⟩ from q) then
      if (State = Req) ∧ (|RSet| < Need) then
        RSet ← RSet ∪ {q}
      else
        send⟨ResT⟩ to q + 1
      end if
    end if
    if (receive⟨PushT⟩ from q) then
      if (Prio ≠⊥) ∧ (State ≠ Req ∨ |RSet| < Need)∧
       (State ≠ In) then
        for all i ∈ RSet do
          send⟨ResT⟩ to i + 1
        end for
        RSet ← ∅
      end if
      send⟨PushT⟩ to q + 1
    end if
    if (receive⟨PrioT⟩ from q) then
      if Prio =⊥ then
        Prio ← q
      else
        send⟨PrioT⟩ to q + 1
      end if
    end if
    if (receive⟨ctrl, C, R, PT, PPr⟩ from q) then
      Ok ← false
      if (q = Succ) ∧ (myC = C) ∧ (Succ ≠ 0) then
        Succ ← Succ + 1
        Ok ← true
        if R then
          RSet ← ∅
          Prio ←⊥
        end if
      end if
      if (q = 0) then
        Ok ← true
        if myC ≠ C then
          Succ ← min(1, Δ_p − 1)
          if R then
            RSet ← ∅
            Prio ←⊥
          end if
        end if
        myC ← C
      end if
      if Ok then
        PT ← min(PT + |RSet|_q, ℓ + 1)
        if Prio = q then
          PPr ← min(PPr + 1, 2)
        end if
        send⟨ctrl, myC, R, PT, PPr⟩ to Succ
      end if
    end if
  end for
  if (State = Req) ∧ (|RSet| ≥ Need) then
    State ← In
    EnterCS()
  end if
  if (State = In) ∧ ReleaseCS() then
    for all i ∈ RSet do
      send⟨ResT⟩ to i + 1
    end for
    RSet ← ∅
    State ← Out
  end if
  if (Prio ≠⊥) ∧ (State ≠ Req ∨ |RSet| ≥ Need) then
    send⟨PrioT⟩ to Prio + 1
    Prio ←⊥
  end if
end repeat
```

variable Prio is set from $\perp$ to the channel number from which the process receives the priority token) until its request is satisfied: the token will then 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.

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 [8] for self-stabilizing ℓ-exclusion protocol on a ring. This mechanism is based on 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* [15] 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 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 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.

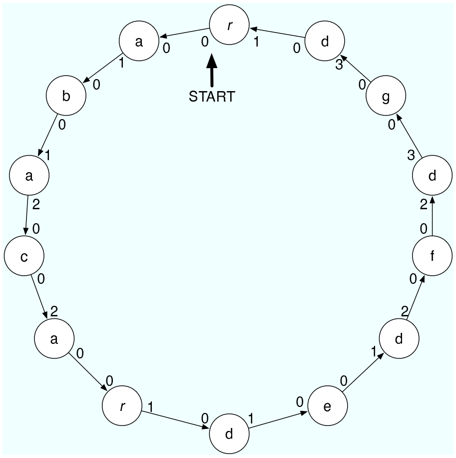

Figure 4. Virtual Ring.

- *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. Then, 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 $PT + \mathtt{SToken}$, 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.

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).[4]

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 they are 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 $\{0 \ldots 2(n-1)(\mathtt{C_{MAX}}+1)\}$, 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 differs 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 $2(n-1)(\mathtt{C_{MAX}}+1)$) 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 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 the `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 $2(n-1)(\mathtt{C_{MAX}}+1)$). In this case, the token marked with

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

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.

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.

Results. For lack of space, the proofs of the two following results have been omitted. The reader can find detailed proofs in the technical report inline at `http://hal.archives-ouvertes.fr/hal-00344193/fr/`.

Theorem 1 *The protocol given in Algorithms 1 and 2 is a self-stabilizing* $\mathtt{k}$*-out-of-*ℓ *exclusion protocol for tree networks.*

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

4. Conclusion and Future Work

In this paper, we propose the first (deterministic) self-stabilizing distributed $\mathtt{k}$-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 channels 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 [7]). However, if we assume unbounded process memory, our solution can be easily adapted to work without assumptions on channels (following the method of [9]).

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 (*e.g*, [1, 4]).

There are several possible extensions of our work. On the theoretical side, one can investigate whether the waiting time of our solution ($\ell \times (2n-3)^2$) 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] Y. Afek and A. Bremler. Self-stabilizing unidirectional network algorithms by power supply. *Chicago Journal of Theoretical Computer Science*, 1998:Article 3, 1998.

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

[3] A. K. 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.

[4] S. Delaët, B. Ducourthial, and S. Tixeuil. Self-stabilization with r-operators revisited. In T. Herman and S. Tixeuil, editors, *Self-Stabilizing Systems*, volume 3764 of *Lecture Notes in Computer Science*, pages 68–80. Springer, 2005.

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

[6] 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.

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

[8] R. Hadid and V. Villain. A new efficient tool for the design of self-stabilizing l-exclusion algorithms: The controller. In A. K. Datta and T. Herman, editors, *WSS*, volume 2194 of *Lecture Notes in Computer Science*, pages 136–151. Springer, 2001.

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

[10] 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.

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

[12] M. Raynal. *Algorithms for Mutual Exclusion*. MIT Press, 1986.

[13] M. Raynal. *Algorithmes du paralllisme, le problme de l'exclusion mutuelle*. Dunod informatique, 1990.

[14] 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.

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