Election in unidirectional rings with homonyms

Karine Altisen, Ajoy K Datta, Stéphane Devismes, Anaïs Durand, Lawrence L Larmore

Election in Unidirectional Rings with Homonyms*

Karine Altisen[a], Ajoy K. Datta[b], Stéphane Devismes[a], Anaïs Durand[a], Lawrence L. Larmore[b]

[a] *Univ. Grenoble Alpes, CNRS, Grenoble INP †, VERIMAG, 38000 Grenoble, France*
[b] *UNLV, Las Vegas, USA*

Abstract

We study leader election in unidirectional rings of homonyms that have no *a priori* knowledge of the number of processes. In this context, we show that there exists no algorithm that solves the process-terminating leader election problem for the class of asymmetrically labeled unidirectional rings. More precisely, we prove that there is no process-terminating leader election algorithm even for the subclass of unidirectional rings where at least one label is unique. Message-terminating leader election is also impossible for the class of unidirectional rings where only a bound on multiplicity is known. However, we show that the process-terminating leader election is possible for two particular subclasses of asymmetrically labeled unidirectional rings where the multiplicity is bounded. We propose three efficient algorithms and analyze their complexities. We also give some non-trivial lower bounds.

Keywords: Leader Election, Homonyms, Multiplicity, Unidirectional Rings

1. Introduction

We consider the *leader election* problem, which consists in distinguishing a unique process of the network. This task is fundamental in distributed systems since it is a basic component in many protocols, *e.g.*, spanning tree construction, broadcasting and convergecasting methods. Leader election is especially helpful to achieve synchronization and self-organization in a network. For example, in a wireless ad-hoc network (WSN), collected data are most of the time aggregated at a leader node, called the *sink*, which can be a gateway between the WSN and other kind of networks.

Leader election is essential, yet sometimes hard to solve, *e.g.*, in 1980, Angluin [1] showed the impossibility of solving deterministic leader election in networks of anonymous processes.

*This work has been partially supported by the ANR projects DESCARTES (ANR-16-CE40-0023) and ESTATE (ANR-16-CE25-0009).

Email addresses: karine.altisen@univ-grenoble-alpes.fr (Karine Altisen), ajoy.datta@unlv.edu (Ajoy K. Datta), stephane.devismes@univ-grenoble-alpes.fr (Stéphane Devismes), anais.durand@univ-grenoble-alpes.fr (Anaïs Durand), lawrence.larmore@unlv.edu (Lawrence L. Larmore)

† Institute of Engineering Univ. Grenoble Alpes

This negative result led to two major opposite lines of research. The first approach circumvents the impossibility result by using randomization to break symmetries [2]. In the second one, networks are assumed to be equipped with unique process identifiers to eliminate symmetries which allowed the design of deterministic algorithms [3]. The notion of *homonym processes* has been introduced as an intermediate model between the (fully) anonymous and (fully) identified ones. In this model, each process has an identifier, called here *label*, which may not be unique. Let $\mathcal{L}$ be the set of labels present in a system of n processes. Then, $|\mathcal{L}| = 1$ (resp., $|\mathcal{L}| = n$) corresponds to the fully anonymous (resp., fully identified) model. This natural extension is motivated by the fact that assuming every process has unique identifier is sometimes too strong in practice. For example, process identifiers often come from MAC addresses and some of these addresses may be duplicated. As a matter of facts, in systems such as *Chord* [4] or *Pastry* [5], addresses are the result of hash functions that are subject to collisions. Moreover, in many cases, system users may wish to preserve some kind of privacy. However, in fully anonymous systems where no identifiers are used, very few problems (including leader election) are deterministically solvable [1, 6, 7, 8]. Consequently, homonymy is an alternative solution to implement trustworthy level of privacy using, for example, *group* or *ring signatures* [9, 10], where signatures are labels and each process of a group share the same signature to anonymously sign messages on behalf of the group. Notice that homonymy using group or ring signatures already have many applications, *e.g.*, in e-voting [11], e-cash [12], and blockchains [13].

Finally, ring topologies, studied here, are the most natural candidates among classical topologies for proof of concept before considering arbitrary ones since they are sparsely connected. Moreover, paying attention to ring networks makes sense from a practical point of view as some real world systems are based on a ring topology, *e.g.*, the token ring standard for local area networks. Ring topologies are also used in P2P systems [14, 15]. For example, *Self-Chord* [15] decouples object keys from peer IDs and sorts keys along a ring.

Related Work. Several recent works [16, 17, 18, 19, 20, 21] studied the leader election problem in networks with homonym processes. Yamashita and Kameda study in [16] the feasibility of leader election in networks of arbitrary topology containing homonym processes. They propose a *process-terminating* (*i.e.*, every process eventually halts) leader election assuming that processes know the size of the network. In [19], Chalopin *et al.* characterize families of (labeled) graphs which admit a process-terminating election algorithm using the notion of quasi-coverings.

In [17], Flocchini *et al.* study the *weak leader election* problem in bidirectional ring networks of homonym processes. In this problem, one or two processes are chosen as leaders. In this latter case, the two elected processes must be neighbors. Under the assumption that processes know *a priori* the number of processes, n, they show that the process-terminating weak leader election is possible if and only if the labeling of the ring is asymmetric, *i.e.*, there is no non-trivial rotational symmetry (*i.e.*, non multiple of n) of the labels resulting in the same labeling. They also propose two process-terminating weak leader election algorithms for asymmetric labeled rings of n processes, assuming that n is prime and that there are only two different labels, 0 and 1. The first algorithm assumes a common sense of direction, *i.e.*, every process is able to distinguish between its clockwise and counterclockwise neighbors. The second algorithm is a generalization of the first one, where the common sense of direction is removed. No time complexity is given for the second algorithm.

In [20], Delporte *et al.* consider the leader election problem in bidirectional ring networks of homonym processes. They propose a necessary and sufficient condition on the number of distinct labels needed to solve the leader election problem. More precisely, they prove that there exists a solution to *message-terminating* (*i.e.*, processes do not halt but only a finite number of messages are exchanged) leader election problem in bidirectional rings if and only if the number of labels is strictly greater than the greatest proper divisor of n. Assuming that condition, they give two algorithms. The first one is message-terminating and does not assume any extra knowledge. On the contrary, the second algorithm is process-terminating but assumes the knowledge of n. They show that their second algorithm is asymptotically optimal in messages ($O(n \log n)$). In [18], Dobrev and Pelc study a generalization of the process-terminating leader election problem in both unidirectional and bidirectional networks of homonym processes. They assume that processes know *a priori* a lower bound m and an upper bound M on the (unknown) number of processes, n. They propose algorithms that decide whether the election is possible and perform it, if so. They propose two synchronous algorithms, one for bidirectional and one for unidirectional rings, and both use $O(M)$ time and $O(n \log n)$ messages. They also propose an asynchronous algorithm for bidirectional rings using $O(nM)$ messages and prove its optimality. No time complexity is given.

In [21], Dereniowski and Pelc study a generalization of the process-terminating leader election in arbitrary networks of homonym processes where processes know *a priori* an upper bound k on the multiplicity of a given label ℓ that exists in the network, *i.e.*, each process knows that ℓ is the label of at least one but at most k processes. They propose a synchronous algorithm that, under these hypotheses, decides whether the election is possible and achieves it, if so. They show that this algorithm is asymptotically optimal in time ($O(k\mathcal{D} + \mathcal{D}\log(n/\mathcal{D}))$, where $\mathcal{D}$ is the diameter of the network).

Contributions. We explore the design and complexity of the (deterministic) process-terminating leader election in unidirectional rings with homonym processes which, contrary to [17, 18, 20], know neither the number of processes n nor any bound on it. Since we only consider *unidirectional* rings, results of Delporte *et al.* [20] do not apply — the common sense of direction may help processes to solve the leader election problem.

We consider five classes of unidirectional labeled rings: $\mathcal{U}^*$, $\mathcal{K}_k$, $\mathcal{A}$, $\mathcal{U}^* \cap \mathcal{K}_k$, and $\mathcal{A} \cap \mathcal{K}_k$. $\mathcal{U}^*$ is the class of all rings in which at least one label is unique. $\mathcal{K}_k$ is the class of all rings where no label occurs more than k times, so k is an *upper bound on the multiplicity* of the labels. Finally, $\mathcal{A}$ is the class of all asymmetric labeled rings, *i.e.*, all labeled rings that have no non-trivial rotational symmetry. By definition, $\mathcal{U}^* \cap \mathcal{K}_k \subset \mathcal{U}^* \subset \mathcal{A}$ and $\mathcal{A} \cap \mathcal{K}_k \subset \mathcal{A}$.

We first establish that the time complexity of any process-terminating leader election algorithm for $\mathcal{U}^* \cap \mathcal{K}_k$ (with $k \geq 2$) is $\Omega(k\ n)$ times units. We then show that any message-terminating leader election algorithm for $\mathcal{U}^* \cap \mathcal{K}_k$ (with $k \geq 2$) requires to exchange $\Omega(kn+n^2)$ bits. By definition, these two lower bounds also hold for $\mathcal{A} \cap \mathcal{K}_k$. Using our lower bound on the time complexity, we derive a simple impossibility result on the process-terminating leader election in $\mathcal{U}^*$, and so in $\mathcal{A}$. Finally, using a direct extension of the impossibility results of Angluin [1], we know that the message-terminating leader election is impossible in $\mathcal{K}_k$ for any $k \geq 2$.

Hence, we focus on the process-terminating leader election in $\mathcal{U}^* \cap \mathcal{K}_k$ and $\mathcal{A} \cap \mathcal{K}_k$ by proposing three algorithms for these classes. The first algorithm, U_k, solves the process-terminating leader election algorithm in $\mathcal{U}^* \cap \mathcal{K}_k$. U_k is asymptotically optimal in time, as its time complexity is $O(k\ n)$ time units. Its message complexity is $O(n^2 + kn)$. Finally, U_k is asymptotically optimal in space, as it requires $O(\log k + b)$ bits per process, where b is the number of bits required to store a label.

Then, we propose two process-terminating leader election algorithms, A_k and B_k, for the more general class $\mathcal{A} \cap \mathcal{K}_k$. Those two algorithms show a trade-off between time and space. A_k is asymptotically optimal in time ($O(k\ n)$), but it requires $O(knb)$ bits per process and $O(kn^2)$ messages are exchanged during an execution. On the contrary, B_k requires only $O(\log k + b)$ bits per process (which is asymptotically optimal), but its time and message complexities are both $O(k^2n^2)$.

Finally, notice that our assumptions on the initial knowledge of processes are not comparable in general with those made in [18, 20]. As a matter of fact, Dobrev and Pelc [18] explain that there are simple cases where the knowledge of some lower bound m and upper bound M on n does not permit their (process-terminating) leader election algorithm to succeed. They illustrate this statement with an example where the initial knowledge of processes is $m = 3$ and $M = 6$. In this case, the 3-node ring with labels 1,2,2 matches those bounds. Now, for this ring their algorithm decides the input is ambiguous and so does not elect a leader because nodes are unable to distinguish whether they are in the 3-node ring with labels 1,2,2 (which is asymmetric), or in a 6-node ring with label 1,2,2,1,2,2 (which is symmetric). We have not such an ambiguity in our settings. Indeed, in our settings (*i.e.*, assuming the knowledge of k and a common orientation), our process-terminating algorithms elect a leader in the 3-node ring with labels 1,2,2 since it (in particular) belongs to $\mathcal{U}^* \cap \mathcal{K}_k$. Similarly, the assumptions made in [20] (*i.e.*, the knowledge of n and the fact that the number of labels is strictly greater than the greatest proper divisor of n) exclude the 6-node ring with label 1,2,2,2,2,2. Indeed, in this labeled ring, the number of labels is 2 and the greatest proper divisor of 6 is 3. Now, in our settings, our algorithms are able to elect a leader in this labeled ring since it belongs (in particular) to $\mathcal{U}^* \cap \mathcal{K}_k$.

Roadmap. Section 2 is dedicated to model and definitions. Lower bounds and impossibility results are presented in Sections 3 and 4. Algorithms U_k, A_k, and B_k are proposed in Sections 5, 6, and 7, respectively, together with their correctness and complexity analysis. We conclude in Section 8 with some perspectives.

Extension of two conference papers. This journal paper is an extension of two preliminary versions, respectively published in the proceedings of SSS'2016 [22] and IPDPS'2017 [23]. Compared to those two conference papers, this journal paper contains significantly new material. Indeed, the lower bounds on the exchanged bit complexity in Section 3 (*i.e.*, Theorem 1 and its associated corollary, Corollary 3) are totally new. Then, concerning impossibility results, Theorem 2 (Section 4) is also new. Concerning now the algorithmic part, Algorithm U_k (Section 5) was originally presented in [22], yet without any proof of correctness or complexity analysis. We have filled all these important blanks in the present paper. Furthermore, it is worth noting that the time and space complexity bounds we prove here are very precise. Finally, we have refined both the correctness proof and complexity analysis of Algorithm B_k initially proposed in [23]. In particular, we have revised the time complexity analysis of Algorithm B_k to obtain tighter bounds.

2. Preliminaries

Ring Networks. We assume unidirectional rings of $n \geq 2$ processes, $p_0, \ldots, p_{n-1}$, operating in asynchronous message-passing model, where links are FIFO and reliable. p_i can only receive messages from its *left* neighbor, p_{i-1}, and can only send messages to its *right* neighbor, p_{i+1}. Subscripts are modulo n. The *state* of a process is a vector of the values of its variables. The state of a link (p_i, p_{i+1}), noted $S_{(p_i,p_{i+1})}$, is the ordered list of messages it contains. A *configuration* is a vector of states, one for each link and each process of the ring. Let γ be a configuration. The state of process p in γ is denoted by $\gamma(p)$. The value of the variable x of p in γ is denoted by $\gamma(p).x$. Processes communicate using the functions **send** and **rcv**. Since every link (p_i, p_{i+1}) is reliable, calls to **send** by p_i and **rcv** by p_{i+1} are the only way to modify $S_{(p_i,p_{i+1})}$. When p_i executes **send** m, the message m is added at the tail of $S_{(p_i,p_{i+1})}$. Let now explain how a call of p_{i+1} to **rcv** works. Each message is of the form $\langle x_1, \ldots, x_k \rangle$, where $x_1, \ldots, x_k$ is a list of values, each of a given datatype. We say that a value x *conforms to* y if y is a value and $x = y$, or y is a variable and has the same datatype as x. A message in $S_{(p_i,p_{i+1})}$ remains in this list until p_{i+1} receives it by calling the function **rcv** (no message loss). The received messages are processed FIFO. So, the function **rcv** is *message-blocking*: A call to **rcv** $\langle v_1, \ldots, v_z \rangle$ by p_{i+1} returns True if and only if the head message $\langle x_1, \ldots, x_z \rangle$ of $S_{(p_i,p_{i+1})}$ satisfies $\forall j \in \{1, \ldots, z\}$, x_j conforms to v_j. When a call to **rcv** $\langle v_1, \ldots, v_z \rangle$ by p_{i+1} returns True, the head of $S_{(p_i,p_{i+1})}$, $\langle x_1, \ldots, x_z \rangle$, is removed from $S_{(p_i,p_{i+1})}$ (each message is received exactly once) and $\forall j \in \{1, \ldots, z\}$, v_j is assigned to x_j if v_j is a variable. Otherwise, **rcv** does not modify $S_{(p_i,p_{i+1})}$.

A distributed algorithm is a collection of n local algorithms, one per process. We assume that processes have no knowledge about n, and each process p has a *label*, $p.id$; labels may not be distinct. For any label ℓ in the ring $\mathcal{R}$, let mlty(ℓ), the *multiplicity* of ℓ in $\mathcal{R}$, be the number of processes in $\mathcal{R}$ whose *id* is ℓ. Comparisons (order and equality) are the only operations permitted on labels. We denote by b the number of bits required to store any label. In our distributed algorithms, all local algorithms are identical, except maybe for the labels. In particular, every execution begins at a so-called *initial configuration*, where each process is at a designated *initial state* and all links are empty. The local algorithm of each process p is given as a list of actions of the form $\langle guard \rangle \to \langle statement \rangle$. A guard is a predicate involving the variables of p and calls to **rcv**. An action is *enabled* if its guard is True. A process p is enabled if at least one of its action is enabled. A statement contains assignments of p's variables and/or calls to the function **send**. The statement of an action can be executed by p only if the action is enabled at p. We assume that the actions are atomically executed, *i.e.*, the evaluation of the guard and the execution of the corresponding statement, if executed, are done in one atomic step. We enforce the local algorithm of each process p to contain at most one action triggerable without the reception of any message. This action should be executed by p first in all executions.

Processes are *fairly activated*, *i.e.*, if a process is continuously enabled, then it eventually executes one of its enabled actions. Let $\mapsto$ be the binary relation over configurations such that $\gamma \mapsto \gamma'$ if and only if γ' can be obtained from γ by the atomic execution of one or more enabled processes in γ; $\gamma \mapsto \gamma'$ is called a *step*. An *execution* is a maximal sequence of configurations $\Gamma = \gamma_0 \ldots \gamma_i \ldots$ such that (1) γ_0 is the initial configuration, (2) $\forall i > 0$, $\gamma_{i-1} \mapsto \gamma_i$, and (3) processes are fairly activated in Γ. *Maximal* means that Γ is either infinite, or ends in a so-called *terminal configuration* where no process is enabled. *Time complexity* [24] is evaluated in *time units*, assuming that message transmission time is at most one time unit, and the process execution time is zero. Roughly speaking, time complexity measures the execution time of the algorithm according to the slowest messages: the execution is normalized in such a way that the longest message delay (*i.e.*, the transmission of the message followed by its processing at the receiving process) becomes one unit of time.

Leader Election. We consider two definitions of the problem of leader election in the message-passing model: the *message-terminating* and the *process-terminating* leader election [24]. Informally, in a process-terminating solution, every process eventually halts, whereas, in a message-terminating solution, processes do not halt but only a finite number of messages is exchanged.

Definition 1 (Message-terminating Leader Election)**.** *An algorithm* ALG *solves the* message-terminating leader election *problem in a ring network $\mathcal{R}$ if every execution e of* ALG *on $\mathcal{R}$ satisfies the following conditions:*

1. *e is finite.*
2. *Each process p has a Boolean variable $p.isLeader$ such that, in the terminal configuration of e, $\ell.isLeader$ is* TRUE *for a unique process ℓ (i.e., the leader).*
3. *Every process p has a variable $p.leader$ such that, in the terminal configuration, $p.leader = \ell.id$, where ℓ satisfies $\ell.isLeader$.*

Definition 2 (Process-terminating Leader Election)**.** *An algorithm* ALG *solves the* process-terminating leader election *problem in a ring network $\mathcal{R}$ if it solves the message-terminating leader election in $\mathcal{R}$ and if every execution e of* ALG *on $\mathcal{R}$ satisfies the following additional conditions:*

4. *For every process p, $p.isLeader$ is initially* FALSE *and never switched from* TRUE *to* FALSE*: each decision of being the leader is irrevocable. Consequently, there should be at most one leader in each configuration.*
5. *Every process p has a Boolean variable $p.done$, initially* FALSE*, such that $p.done$ is eventually* TRUE *for all p, indicating that p knows that the leader has been elected. More precisely, once $p.done$ becomes* TRUE*, it will never become* FALSE *again, $\ell.isLeader$ is equal to* TRUE *for a unique process ℓ, and $p.leader$ is permanently set to $\ell.id$.*

6. *Every process p eventually* halts*, i.e., locally decides its termination, after $p.done$ becomes* TRUE.

Ring Networks Classes. An algorithm ALG solves the message-terminating (resp. process-terminating) leader election *for the class of ring networks C* if it solves the message-terminating (resp. process-terminating) leader election for every ring network $\mathcal{R} \in C$. In particular, ALG cannot be given any specific information about the network (such as its cardinality or the actual multiplicity of labels) unless that information holds for all ring networks of C. Indeed, ALG must work for every $\mathcal{R} \in C$ without any change whatsoever in its code.

A ring network $\mathcal{R}$ of n processes is said to be *symmetric* if a non-trivial rotation of the labels results in the same labeling, *i.e.*, there is some integer $0 < d < n$ such that, for all $i \geq 0$, p_i and p_{i+d} have the same label. Otherwise, $\mathcal{R}$ is said to be *asymmetric*.

We mainly consider the three following classes of ring networks.

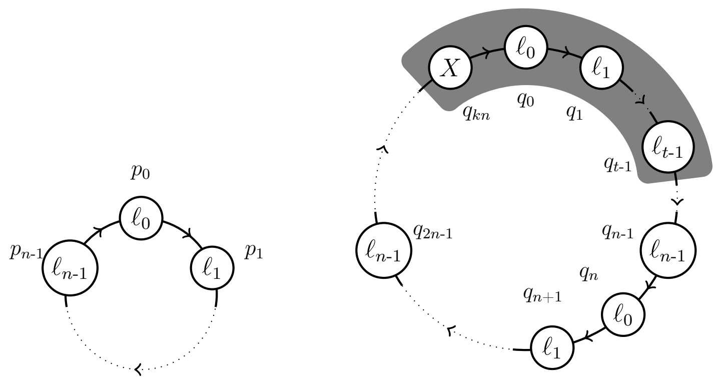
- $\mathcal{A}$ is the class of all asymmetric unidirectional ring networks.
- $\mathcal{U}^*$ is the class of all unidirectional ring networks in which at least one process has a unique label. By definition, $\mathcal{U}^* \subset \mathcal{A}$.
- $\mathcal{K}_k$, with $k \geq 1$ a given integer, is the class of all unidirectional ring networks where no more than k processes have the same label: k is an upper bound on the multiplicity of labels in $\mathcal{R} \in \mathcal{K}_k$. Notice that $\mathcal{K}_1 \subset \mathcal{U}^*$ and $\mathcal{K}_1 \subset \mathcal{K}_2 \ldots$

3. Lower Bounds

We first establish a lower bound that depends on k on the execution time of any process-terminating leader election algorithm in $\mathcal{U}^* \cap \mathcal{K}_k$ (with $k \geq 2$). Even though, it has been written independently, it appears that the proof of the technical result below is essentially an adaptation of the proof of the "Quasi-Lifting corollary" [19] to our context.

Lemma 1. *Let $k \geq 2$ and* ALG *be an algorithm that solves the process-terminating leader election for $\mathcal{U}^* \cap \mathcal{K}_k$. $\forall \mathcal{R} \in \mathcal{K}_1$, the synchronous execution of* ALG *in $\mathcal{R}$ lasts at least $1 + (k-2)\,n$ time units, where n is the number of processes.*

Proof. Let $k \geq 2$ and ALG be a process-terminating leader election algorithm for $\mathcal{U}^* \cap \mathcal{K}_k$. Let $\mathcal{R}_n \in \mathcal{K}_1$ be a ring of n processes, noted $p_0, \ldots, p_{n-1}$ with distinct labels $\ell_0, \ldots, \ell_{n-1}$ respectively, see Figure 1a. Since $\mathcal{K}_1 \subseteq \mathcal{U}^* \cap \mathcal{K}_k$, ALG is correct for $\mathcal{R}_n$ and so, the synchronous execution $e = (\gamma_i)_{i \geq 0}$ of ALG on $\mathcal{R}_n$ is finite and a process is elected. Let T be the execution time of e: within T time units in e, $p_L.isLeader$ becomes TRUE for some $0 \leq L \leq n-1$, *i.e.*, p_L is the leader in the terminal configuration γ_T of e. We now build the ring $\mathcal{R}_{n,k} \in \mathcal{U}^* \cap \mathcal{K}_k$ of $kn+1$ processes, $q_0, \ldots, q_{kn}$, with labels consisting of the sequence $\ell_0, \ldots, \ell_{n-1}$ repeated k times, followed by a single label $X \notin \{\ell_0, \ldots, \ell_{n-1}\}$, see Figure 1b. Let $e' = (\gamma'_i)_{i \geq 0}$ be the synchronous execution of ALG on $\mathcal{R}_{n,k}$. Since $\mathcal{R}_{n,k} \in \mathcal{U}^* \cap \mathcal{K}_k$, ALG is correct on $\mathcal{R}_{n,k}$ so e' is finite and there is no configuration along e' such that two processes declare themselves leader. By construction, after $t \geq 0$ time units, only the processes q_i, with $i \in \{0, \ldots, t-1\}$, can have received information from process q_{kn} of label X, see the gray zone on Figure 1b.



(a) Ring $\mathcal{R}_n$ (b) Ring $\mathcal{R}_{n,k}$

Figure 1: Illustration of the proof of Lemma 1. In gray, the processes of $\mathcal{R}_{n,k}$ that can have received information from q_{kn} of label X within $t \geq 0$ time units.

Hence, we have the following property on e': $(*)$ For every $j \in \{0, ..., kn-1\}$, for every $t \geq 0$, if $t \leq j$, then the state of q_j in γ'_t is identical to the state of $p_{j \bmod n}$ in γ_t.

Assume, by contradiction, that $T \leq (k-2)n$. Let $j_1 = (k-2)n+L$ and $j_2 = (k-1)n+L$. Since $L \in \{0, ..., n-1\}$, we have $j_1, j_2 \in \{0, ..., kn-1\}$, hence $T \leq j_1 < j_2$. Moreover, $j_1 \bmod n = j_2 \bmod n = L$. So, by $(*)$ the states of q_{j_1} and q_{j_2} in γ'_T are identical to the state of p_L in γ_T: in particular, $\gamma'_T(q_{j_1}).\mathit{isLeader} = \gamma'_T(q_{j_2}).\mathit{isLeader} = \textsc{True}$. This contradicts the fact that Alg is a process-terminating leader election algorithm for $R_{n,k}$. (Bullet 5 of the specification is violated in γ'_T, see p. 6.) Hence, the execution time T of the synchronous execution of Alg in R_n is greater than $(k-2)n$. □

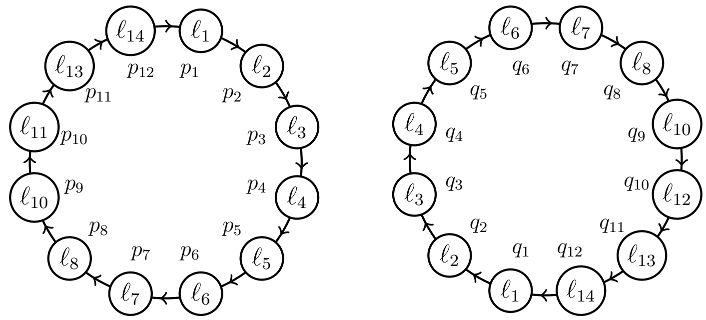
Since $\mathcal{K}_1 \subseteq \mathcal{U}^* \cap \mathcal{K}_k$, follows:

Corollary 1. *Let $k \geq 2$. The time complexity of any algorithm that solves the process-terminating leader election for $\mathcal{U}^* \cap \mathcal{K}_k$ is $\Omega(k\,n)$ time units, where n is the number of processes.*

Furthermore, by definition $\mathcal{U}^* \subseteq \mathcal{A}$, and so:

Corollary 2. *Let $k \geq 2$. The time complexity of any algorithm that solves the process-terminating leader election for $\mathcal{A} \cap \mathcal{K}_k$ is $\Omega(k\,n)$ time units, where n is the number of processes.*

The algorithm proposed by Peterson [25] to solve leader election in identified ring networks has message complexity $O(n \log n)$ and each message contains $\Theta(b)$ bits, *i.e.*, the amount of exchanged information is $O(b\ n \log n)$. As commonly done in the literature, we can assume that $b = \Theta(\log n)$ in identified networks, so $O(b\ n \log n) = O(n \log^2 n)$. Then, as the algorithm of Peterson applies for $\mathcal{U}^* \cap \mathcal{K}_1$, we might expect that there exists a leader election algorithm for class $\mathcal{U}^* \cap \mathcal{K}_k$ whose required amount of exchanged information is $O(k\ n\ \log^2 n)$. Theorem 1 shows that, when k is fixed, the minimum amount of exchanged bits needed to solve leader election in the worst case is greater than what we might expect when n is large.



(a) $\mathcal{R}_I$ (b) $\mathcal{R}_J$

Figure 2: Ring networks $\mathcal{R}_I$ and $\mathcal{R}_J$ where $m = 7$, $I = \ell_8, \ell_{10}, \ell_{11}, \ell_{13}, \ell_{14}$, and $J = \ell_8, \ell_{10}, \ell_{12}, \ell_{13}, \ell_{14}$ used in the proof of Theorem 1.

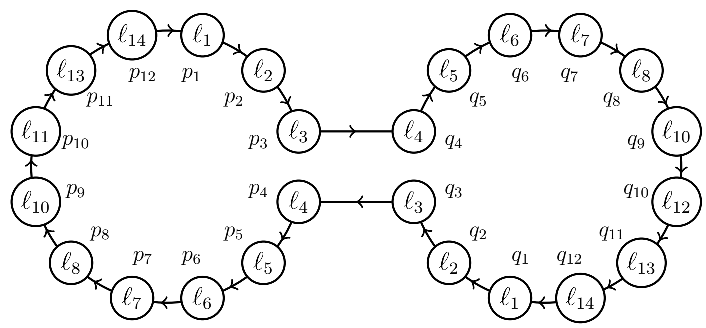


Figure 3: Ring network $\mathcal{R}^p_{I\#J}$ where $m = 7$, $p = 3$, $I = \ell_8, \ell_{10}, \ell_{11}, \ell_{13}, \ell_{14}$, and $J = \ell_8, \ell_{10}, \ell_{12}, \ell_{13}, \ell_{14}$ used in the proof of Theorem 1.

Theorem 1. *Let $k \geq 2$. For any message-terminating leader election algorithm* Alg *for $\mathcal{U}^* \cap \mathcal{K}_k$, there exists executions of* Alg *during which $\Omega(kn + n^2)$ bits are exchanged, where n is the number of processes.*

Proof. Let $k \geq 2$. Let Alg be a message-terminating leader election algorithm for $\mathcal{U}^* \cap \mathcal{K}_k$. Let $m \geq 2$ and let $n = 2m$. Let $\ell_1, \ldots, \ell_n$ be distinct labels. Let $\mathcal{L}$ be the set of non-empty proper subsequences of $(\ell_{m+1}, \ldots, \ell_n)$, *i.e.*, the subsequences of $(\ell_{m+1}, \ldots, \ell_n)$ whose length is at least one but at most $m-1$. For any $I \in \mathcal{L}$, $\mathcal{R}_I$ is the ring network containing $m + |I|$ processes, denoted $p_1, p_2, \ldots, p_m, p_{m+1}, \ldots, p_{m+|I|}$, whose label sequence is $\Lambda_I = \ell_1, \ell_2, \ldots, \ell_m, I$. More precisely, in $\mathcal{R}_I$, for every $i \in \{1, \ldots, m\}$, $p_i.id = \ell_i$ and for every $j \in \{1, \ldots, |I|\}$, $p_{m+j}.id = I[j]$ (the j^{th} element of I). The ring network $\mathcal{R}_I$ for $m = 7$ and $I = \ell_8, \ell_{10}, \ell_{11}, \ell_{13}, \ell_{14}$ is illustrated on Figure 2a.

Let $\mathfrak{R} = \{\mathcal{R}_I : I \in \mathcal{L}\}$. Notice that $|\mathfrak{R}| = 2^m - 2$ (since the empty sequence and $\{\ell_{m+1}, \ldots, \ell_n\}$ are not in $\mathcal{L}$). Furthermore, every label in $\mathcal{R}_I$ is unique so $\mathcal{R}_I \in \mathcal{U}^* \cap \mathcal{K}_k$. Hence, Alg is correct for every $\mathcal{R}_I \in \mathfrak{R}$. For every $I, J \in \mathcal{L}$, let $\mathcal{R}_{I\#J}$ be the ring network containing $2m + |I| + |J|$ processes whose label sequence is $\Lambda_I \Lambda_J$. $\mathcal{R}_{I\#J}$ can be obtained from $\mathcal{R}_I$ and $\mathcal{R}_J$ as follows (we denote by $q_1, \ldots, q_{m+|J|}$ the processes of $\mathcal{R}_J$ to avoid confusion). For some $p \in \{1, \ldots, m-1\}$, we obtain $\mathcal{R}^p_{I\#J}$ when we join $\mathcal{R}_I$ and $\mathcal{R}_J$ by removing edges (p_p, p_{p+1}) and (q_p, q_{p+1}) and replacing them by edges (p_p, q_{p+1}) and (q_p, p_{p+1}). Figure 3 shows the ring network $\mathcal{R}^p_{I\#J}$ for $p = 3$, $I = \ell_8, \ell_{10}, \ell_{11}, \ell_{13}, \ell_{14}$, and $J = \ell_8, \ell_{10}, \ell_{12}, \ell_{13}, \ell_{14}$ obtained by joining $\mathcal{R}_I$ (see Figure 2a) and $\mathcal{R}_J$ (see Figure 2b).

Claim 1: For every $p \in \{1, \ldots, m-1\}$, if $I \neq J$, then $\mathcal{R}^p_{I\#J} \in \mathcal{U}^* \cap \mathcal{K}_k$.

Proof of Claim 1: No label appears more than twice in $\mathcal{R}^p_{I\#J}$ so $\mathcal{R}^p_{I\#J} \in \mathcal{K}_k$. Then, without loss of generality, assume $|I| \leq |J|$. So, there is some $\ell_j \in J \backslash I$ and ℓ_j is a unique label in $\mathcal{R}^p_{I\#J}$. Hence, $\mathcal{R}^p_{I\#J} \in \mathcal{U}^*$. ∎

For every $I \in \mathcal{L}$, let e_I be the synchronous execution of Alg on $\mathcal{R}_I$. For each $p \in \{1, \ldots, m\}$, let $\sigma_{I,p}$ be the stream (sequence) of bits sent by p_p to p_{p+1} during e_I.

Claim 2: For any $I, J \in \mathcal{I}$ and any $p \in \{1, \ldots, m-1\}$, if $\sigma_{I,p} = \sigma_{J,p}$, then $I = J$.

Proof of Claim 2: Let $I, J \in \mathcal{I}$ and $p \in \{1, \ldots, m-1\}$. Assume that $\sigma_{I,p} = \sigma_{J,p}$. Let $\mathcal{R}^p_{I\#J}$ the ring network obtained by joining $\mathcal{R}_I$ and $\mathcal{R}_J$ at the edges (p_p, p_{p+1}) and (q_p, q_{p+1}). Let $e^p_{I\#J}$ be the synchronous execution of Alg on $\mathcal{R}^p_{I\#J}$.

First, we show by induction on the steps of $e^p_{I\#J}$ that $\forall x \geq 1$, every process p_i, $i \in \{1, \ldots, m+|I|\}$ (respectively, q_j, $j \in \{1, \ldots, m+|J|\}$) sends the same bits during the x^{th} step of $e^p_{I\#J}$ than in the x^{th} step of e_I (respectively, e_J).

Base Case: Alg is a deterministic algorithm and every process p_i (respectively, q_j) has the same initial state (in particular, the same label) in $e^p_{I\#J}$ than in e_I (respectively, e_J). Hence every process p_i (respectively, q_j) sends the same bits during the first step of $e^p_{I\#J}$ than during the first step of e_I (respectively, e_J).

Induction Step: Assume that every process p_i, $i \in \{1, \ldots, m+|I|\}$ (respectively, q_j, $j \in \{1, \ldots, m+|J|\}$) sends the same bits during the x^{th} step of $e^p_{I\#J}$ than in the x^{th} step of e_I (respectively, e_J), $x \geq 1$. Consider the $(x+1)^{\text{th}}$ step of $e^p_{I\#J}$.

Every process p_i, $i \in \{1, \ldots, p\} \cup \{p+2, m+|I|\}$, (respectively, q_j, $j \in \{1, \ldots, p\} \cup \{p+2, \ldots, m+|J|\}$) has the same predecessor in $\mathcal{R}^p_{I\#J}$ than in $\mathcal{R}_I$ (respectively, $\mathcal{R}_J$). By induction hypothesis, this predecessor sends the same bits during the x^{th} step of $e^p_{I\#J}$ than during the x^{th} step of e_I (respectively, e_J).

Now, the only processes that do not have the same predecessor in $\mathcal{R}^p_{I\#J}$ than in $\mathcal{R}_I$ or $\mathcal{R}_J$ are p_{p+1} and q_{p+1}. By induction hypothesis, their predecessor, respectively q_p and p_p, send the same bits during the x^{th} step of respectively $e^p_{I\#J}$ than during the x^{th} step of respectively e_J and e_I. Furthermore, $\sigma_{I,p} = \sigma_{J,p}$ so they send the exact same bits.

Hence, every process receives the same bits and so send the same bits (since the algorithm is deterministic) during the $(x+1)^{\text{th}}$ step of $e^p_{I\#J}$ than during the $(x+1)^{\text{th}}$ step of e_I or e_J.

Hence, the processes cannot distinguish $e^p_{I\#J}$ and e_I or e_J. So both the process that declares itself leader in e_I and the one that declares itself leader in e_J also declares itself leader in $e^p_{I\#J}$. Now, assume by contradiction that $I \neq J$. By Claim 1, $\mathcal{R}^p_{I\#J} \in \mathcal{U}^* \cap \mathcal{K}_k$. Since two processes declare themselves leader in $e^p_{I\#J}$, we have a contradiction with the correctness of Alg for class $\mathcal{U}^* \cap \mathcal{K}_k$. ∎

The rest of the proof is based on a counting argument, *i.e.*, there are not enough bit streams to distinguish the rings of $\mathfrak{R}$, unless those streams have length $\Omega(n^2)$. Let $a = m-2$. Recall that a set of cardinality m has 2^m subsets including $2^m - 2$ non-empty proper subsets.

The number of bit streams of length at most a is:

$$\sum_{l=1}^{a} 2^l = 2^{a+1} - 1 = 2^{m-1} - 1 < 2^m - 2$$

Let $p \in \{1, \ldots, m-1\}$. Let $\mathcal{L}_p = \{I \in \mathcal{L} : |\sigma_{I,p}| \leq a\}$. By Claim 2, $\mathcal{L}_p$ has cardinality less than $2^m - 2$, so there exists some $I \in \mathcal{L}$ that is not a member of $\mathcal{L}_p$, for any $p \in \{1, \ldots, m-1\}$. Hence, $\Omega(m\ a) = \Omega(n^2)$ bits are exchanged during the synchronous execution of $\textsc{Alg}$ on $\mathcal{R}_I$. Now, at least one message must be exchanged at each step, so, by Corollary 1, there exists executions of $\textsc{Alg}$ where $\Omega(kn + n^2)$ bits are exchanged. $\square$

Since $\mathcal{U}^* \cap \mathcal{K}_k \subseteq \mathcal{A} \cap \mathcal{K}_k$, follows:

Corollary 3. *Let $k \geq 2$. For any message-terminating leader election algorithm* $\textsc{Alg}$ *for $\mathcal{A} \cap \mathcal{K}_k$, there exists executions of* $\textsc{Alg}$ *during which $\Omega(kn + n^2)$ bits are exchanged, where n is the number of processes.*

4. Impossibility Results

Class $\mathcal{K}_k$. Below, we extend the impossibility results of Angluin [1] to $\mathcal{K}_k$.

Theorem 2. *There is no algorithm that solves message-terminating leader election in a symmetric ring of at least 2 processes.*

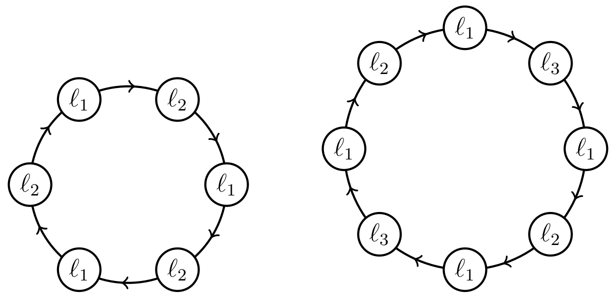
Proof. Let $\mathcal{R}$ be a symmetric ring of $n \geq 2$ processes. Let $0 < d < n$ such that, for all $i \geq 0$, p_i and p_{i+d} have the same label. Assume by contradiction that $\textsc{Alg}$ is a message-terminating leader election algorithm for $\mathcal{R}$. Let $e = (\gamma_j)_{j \geq 0}$ be the synchronous execution of $\textsc{Alg}$ on $\mathcal{R}$. At every step of e, each p_i, $i \geq 0$, makes exactly the same actions as p_{i+d}, and thus, every configuration of e is symmetric; *i.e.*, for all $1 \leq i \leq n$ and for all configurations γ_j, $j \geq 0$, of e, all variables of p_i and p_{i+d} have the same value. Eventually, a terminal configuration γ_T is reached. Let p_ℓ be the elected leader in γ_T; thus $\gamma_T(p_\ell).isLeader = \textsc{True}$. But $\gamma_T(p_{\ell+d}).isLeader$ also, which contradicts the uniqueness of the leader in a solution, since $p_{\ell+d} \neq p_\ell$. $\square$

Class $\mathcal{K}_k$, $k \geq 2$, contains symmetric rings of at least two processes, *e.g.*, see Figure 4. Hence, we have:

Theorem 3. *For any $k \geq 2$, there is no algorithm that solves message-terminating leader election for $\mathcal{K}_k$.*

Classes $\mathcal{U}^*$ and $\mathcal{A}$. Using Lemma 1, we can easily derive the following two impossibility results.

Theorem 4. *There is no algorithm that solves the process-terminating leader election for $\mathcal{U}^*$.*



(a) Ring in $\mathcal{K}_3$. (b) Ring in $\mathcal{K}_4$.

Figure 4: Examples of symmetric ring networks in $\mathcal{K}_k$.

Proof. Suppose ALG is an algorithm for $\mathcal{U}^*$. Let $\mathcal{R}_n$ be a ring network of $\mathcal{K}_1$ with n processes. Let e be the synchronous execution of ALG on $\mathcal{R}_n$: as $\mathcal{K}_1 \subseteq \mathcal{U}^*$, ALG is correct for $\mathcal{R}_n$ and, consequently, e is finite. Let T be the number of steps of e. We can fix some $k \geq 2$ such that $1 + (k-2)\,n > T$.

Since $(\mathcal{U}^* \cap \mathcal{K}_k) \subseteq \mathcal{U}^*$, ALG is correct for $\mathcal{U}^* \cap \mathcal{K}_k$. By Lemma 1, $T \geq 1 + (k-2)n$, a contradiction. □

Since by definition $\mathcal{U}^* \subseteq \mathcal{A}$, Theorem 4 implies the following corollary.

Corollary 4. *There is no algorithm that solves the process-terminating leader election for $\mathcal{A}$.*

5. Algorithm U_k

In this section, we present Algorithm U_k which solves the process-terminating leader election for the class $\mathcal{U}^* \cap \mathcal{K}_k$, with fixed $k \geq 1$ (see Algorithm 1).

5.1. Variables of U_k

U_k elects the process of minimum unique label to be the leader, namely the process L such that $L.id = \min\{p.id : p \in V \wedge \text{mlty}(p.id) = 1\}$. In U_k, each process p has the following variables.

1. $p.id$, (constant) input of unspecified *label type*, the label of p.
2. $p.init$, Boolean, initially TRUE.
3. $p.active$, Boolean, which indicates that p is *active*. If $\neg p.active$, we say p is *passive.* Initially, all processes are active, and when U_k is done, the leader is the only active process. A passive process never becomes active.
4. $p.count$, an integer in the range $0 \ldots k+1$. Initially, $p.count = 0$. $p.count$ will give to p a rough estimate of the frequency of its label in the ring.
5. $p.leader$, of label type. When U_k is done, $p.leader = L.id$.
6. $p.isLeader$, Boolean, initially FALSE, follows the problem specification: eventually, $L.isLeader$ becomes TRUE and remains TRUE, while, for all $p \neq L$, $p.isLeader$ remains FALSE for the entire execution.

Algorithm 1: Actions of Process p in Algorithm U_k.

A1	::	$p.init$	$\rightarrow$	$p.init \leftarrow$ FALSE $\textbf{send}\ \langle p.id, 0\rangle$
A2	::	$\neg p.init \wedge p.active \wedge \textbf{rcv}\ \langle x, c\rangle \wedge x \neq p.id$ $\wedge\ (p.count = 0 \vee c > p.count)$	$\rightarrow$	$\textbf{send}\ \langle x, c\rangle$
A3	::	$\neg p.init \wedge p.active \wedge \textbf{rcv}\ \langle x, c\rangle \wedge x > p.id$ $\wedge\ c = p.count \wedge c \geq 1$	$\rightarrow$	$\textbf{send}\ \langle x, c\rangle$
A4	::	$\neg p.init \wedge p.active \wedge \textbf{rcv}\ \langle x, c\rangle \wedge x = p.id$ $\wedge\ c = p.count \wedge c \leq k-1$	$\rightarrow$	$p.count \leftarrow c+1$ $\textbf{send}\ \langle x, c+1\rangle$
(Deactivation)				
A5	::	$\neg p.init \wedge p.active \wedge \textbf{rcv}\ \langle x, c\rangle \wedge x \neq p.id$ $\wedge\ c < p.count$	$\rightarrow$	$p.active \leftarrow$ FALSE $\textbf{send}\ \langle x, c\rangle$
A6	::	$\neg p.init \wedge p.active \wedge \textbf{rcv}\ \langle x, c\rangle \wedge x < p.id$ $\wedge\ c = p.count \wedge c \geq 1$	$\rightarrow$	$p.active \leftarrow$ FALSE $\textbf{send}\ \langle x, c\rangle$
(Passive Processes)				
A7	::	$\neg p.init \wedge \neg p.active \wedge \textbf{rcv}\ \langle x, c\rangle \wedge x \neq p.id \wedge c \leq k$	$\rightarrow$	$\textbf{send}\ \langle x, c\rangle$
A8	::	$\neg p.init \wedge \neg p.active \wedge \textbf{rcv}\ \langle x, c\rangle \wedge x = p.id$	$\rightarrow$	(nothing)
(Termination)				
A9	::	$\neg p.init \wedge p.active \wedge \textbf{rcv}\ \langle x, k\rangle \wedge x = p.id$ $\wedge\ p.count = k$	$\rightarrow$	$p.isLeader \leftarrow$ TRUE $p.leader \leftarrow p.id$ $p.done \leftarrow$ TRUE $p.count \leftarrow k+1$ $\textbf{send}\ \langle x, k+1\rangle$
A10	::	$\neg p.init \wedge \neg p.active \wedge \textbf{rcv}\ \langle x, k+1\rangle$	$\rightarrow$	$p.leader \leftarrow x$ $p.done \leftarrow$ TRUE $\textbf{send}\ \langle x, k+1\rangle$ (halt)
A11	::	$\neg p.init \wedge p.active \wedge \textbf{rcv}\ \langle x, k+1\rangle \wedge x = p.id$ $\wedge\ p.count = k+1$	$\rightarrow$	(halt)

7. $p.done$, Boolean, initially FALSE, follows the problem specification: eventually, $p.done =$ TRUE for all p. $p.done$ means that p knows a leader has been elected; once true, it will never become false.

5.2. Messages of U_k

U_k uses only one kind of message. Each message is the forwarding of a *token* which is generated at the initialization of the algorithm, and is of the form $\langle x, c\rangle$, where x is the label of the originating process, and c is a *counter*, an integer in the range $0 \ldots k+1$, initially zero.

5.3. Overview of U_k

The explanation below is illustrated by the example in Figure 5. The fundamental idea of U_k is that a process becomes passive, *i.e.*, is no more candidate for the election, if it receives a token that proves its label is not unique or is not the smallest unique label.

Counter Increments. Initially, every process initiates a token with its own label and counter zero (see (a)). No tokens are initiated afterwards. Each token moves around the ring clockwise – every time it is forwarded, its counter and the local counter of the processes are

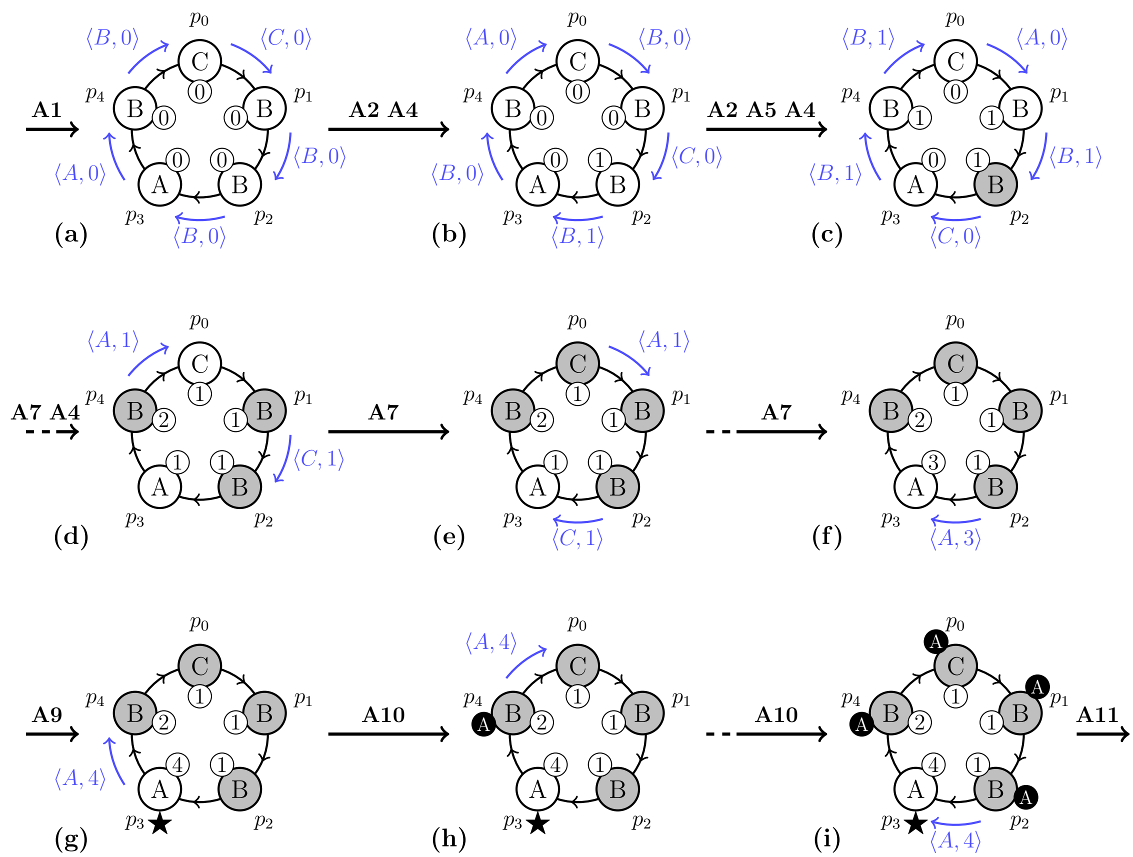


Figure 5: Example of execution of U_k where $k = 3$. The counter of a process is in the white bubble next to the corresponding node. Gray nodes are passive. $p.isLeader = \text{TRUE}$ if there is a star next to the node. The black bubble contains the elected label, $p.leader$.

incremented if the forwarding process has the same label as the token (*e.g.*, Step (a)$\mapsto$(b)). Thus, if the message $\langle x, c\rangle$ is in a channel, that token was initiated by a process whose label is x, and has been forwarded c times by processes whose labels are also x. The token could also have been forwarded any number of times by processes with labels which are not x. Thus, the counter in a token is a rough estimate of the frequency of its label in the ring.

Non-unique Label Elimination. If an active process p receives a message tagged with a label different from $p.id$ whose counter is (strictly) less than $p.count$, this proves the label of p is not unique since the counter $p.count$ grows faster than the one of another label. In this case, p executes **A5**-action and becomes passive (*e.g.*, Step (b)$\mapsto$(c)). Since the counter in the token initiated by L is never incremented, except by L itself, L cannot become passive using this rule. Moreover, every process whose label is not unique becomes passive during the first two ring traversals of the token initiated by L.

Non-lowest Unique Label Elimination. Similarly, if an active process p has a unique label but not the smallest one, it will become passive executing **A6**-action when p receives a message with the same non-zero counter but a label lower than $p.id$ (*e.g.*, Step (d)$\mapsto$(e)). This happens at the latest when the process receives the message $\langle L.id, 1\rangle$, *i.e.*, before the second time L receives its own token. So, after the token of L has made two traversals of the ring, every process but L is passive. Moreover, the token initiated by L is the only surviving token because all other tokens have vanished using **A8**-action.

Termination Detection. The execution continues until the leader L has seen its own label return to it $k+1$ times (*i.e.*, until L receives $\langle L.id, k\rangle$ since the counter inside its token is initialized to zero), otherwise L cannot be sure that what it has seen is not part of a larger ring instead of several rounds of a small ring. Then, L designates itself as leader by **A9**-action (see Step (f)$\mapsto$(g)) and its token does a last traversal of the ring to inform the other processes of its election (*e.g.*, Step (g)$\mapsto$(h)). The execution ends when L receives its token after $k+2$ traversals (see (i)).

5.4. Correctness and Complexity Analysis

To prove the correctness of U_k (Theorem 5), we first prove some results on the counters inside the tokens (Lemma 2). Then, Lemmas 3-7 prove properties on the different phases of U_k. Finally, Theorem 6 gives a complexity analysis of U_k.

In the following proofs, we write #hop(m) for the number of hops, made so far by the token associated to the message m. Notice that #hop(m) is always of the form $an+b$ where $a \geq 0$ is the number of complete traversals realized by the token and $0 \leq b < n$ is the (clockwise) hop-distance from the initiator of the token to its last tokenholder.

Lemma 2. *Let $\gamma \mapsto \gamma'$ be a step. Suppose a message $\langle x, c\rangle$ such that $\#hop(\langle x, c\rangle) = an + b$ in γ with $a \geq 0$ and $0 \leq b < n$ is sent in $\gamma \mapsto \gamma'$. Then:*

1. *$c \geq a$,*
2. *if x is a unique label, then $c = a$, and*
3. *if x is a not a unique label and $a \geq 1$, then $c > a$.*

Proof. Let p be the process which originated the token currently carried by the message m. The token has made a complete traversals of the ring, and has visited p a times, hence its counter has been incremented at least a times. This proves 1. If p is the only process with label x, then the counter has not otherwise been incremented, and we have 2. Suppose x is not a unique label, and $a \geq 1$. There are at least two processes with label x. The token has made at least a full traversals, and thus has been sent by processes of label x at least $2a$ times. Starting at zero, c has been incremented at least $2a$ times, hence $c \geq 2a > a$, and we have 3. □

For the next lemma, we recall that a process can become passive only by executing **A5** or **A6**-action.

Lemma 3. *L never becomes passive.*

Proof. By contradiction, assume L becomes passive during some step $\gamma \mapsto \gamma'$. Then L executes **A5** or **A6**-action, receiving the message $\langle x, c\rangle$ for some $x \neq L.id$. Since the label of L is unique, the token it initiated is still circulating in the ring in γ (it cannot be discarded except by L if it is passive). Moreover, since $x \neq L.id$, $\#\text{hop}(\langle x, c\rangle)$ is not a multiple of n in γ. Let $\#\text{hop}(\langle x, c\rangle) = an + b$ in γ, where $a \geq 0$ and $1 \leq b < n$. Since the links are FIFO, the token initiated by L has made a full circuits during the prefix of execution leading to γ, and $\gamma(L).count = a$. We now consider two cases.

- **Case 1:** x is a unique label. By Lemma 2.2, $c = a = L.count$. Thus, L cannot execute **A5**-action, and since $L.id < x$, L cannot execute **A6**-action either, a contradiction.
- **Case 2:** x is not unique. (Recall that $L.count = a$ in γ.) If $a = 0$, then L is not enabled to execute either action. If $a \geq 1$, then $c > a$ by Lemma 2.3, contradiction.

□

We define an L-tour as follows. Let $e = (\gamma_i)_{i \geq 0}$ be an execution of U_k. The first L-tour of e is the minimum prefix $\gamma_0 \dots \gamma_j$ of e such that L receives (and treats) a message tagged with its own label (for the first time) in step $\gamma_{j-1} \mapsto \gamma_j$. If γ_j is not a terminal configuration, then the second L-tour is the first L-tour of the execution suffix $e' = (\gamma_i)_{i \geq j}$ starting in γ_j, and so forth. From Lemma 3, the code of the algorithm, and the fact that the label of L is unique, we have:

Corollary 5. *Any execution contains exactly $k + 2$ complete L-tours.*

Lemma 4. *For any process p, if $p \neq L$ and $p.id$ is a unique label, then p becomes passive within the first two L-tours.*

Proof. Let $x = L.id$. By definition of L, $x < p.id$. Let $d = \|L, p\|$. Suppose by contradiction that p does not become passive during the first two L-tours (which are defined, Corollary 5). The token t initiated by L is received by p during the first (resp. second) L-tour while $\#\text{hop}(t) = d$ (resp. $\#\text{hop}(t) = n + d$). p receives the token it initiates exactly once before receiving the token $t = \langle x, c\rangle$ (initiated by L) during the second L-tour, say in step $\gamma \mapsto \gamma'$. So, as $p.id$ is unique, we have $p.count = 1$ in γ. Now, $c = 1$ in γ (Lemma 2.1). Thus, p becomes passive by executing **A6**-action in $\gamma \mapsto \gamma'$, contradiction. □

Lemma 5. *If z is a non-unique label, then all processes of label z become passive within the first two L-tours.*

Proof. Let $m \geq 2$ be the multiplicity of z, and let $\mathcal{P}[z] = \{x_1, x_2, \dots x_m\}$ be the sequence of processes of label z in clockwise order from L.

Claim 1: Any process x_i with $i \neq 1$ receives the token initiated by x_{i-1} of the form $\langle z, 0\rangle$ during the first L-tour before receiving $\langle L.id, 0\rangle$.

Proof of Claim 1: L is not between x_{i-1} and x_i, and no process between x_{i-1} and x_i can stop the message $\langle z, 0\rangle$ initiated by x_{i-1}. So, x_i will receive $\langle z, 0\rangle$ before receiving $\langle L.id, 0\rangle$ during the first L-tour. ■

Claim 2: x_1 receives $\langle z, 0\rangle$ and then $\langle z, 1\rangle$ during the first two L-tours, both of them before receiving $\langle L.id, 1\rangle$.

Proof of Claim 2: No process between x_m and x_1 can stop the message $\langle z, 0\rangle$ initiated by x_m. Then, by Claim 1, x_m receives a message $\langle z, 0\rangle$, while satisfying $x_m.count = 0$. So, x_m sends $\langle z, 1\rangle$ after $\langle z, 0\rangle$, but before receiving $\langle L.id, 0\rangle$. Again, no process between x_m and x_1 can stop that message. So, x_1 receives $\langle z, 0\rangle$ and $\langle z, 1\rangle$ before receiving $\langle L.id, 1\rangle$, *i.e.*, during the first two L-tours. ■

Claim 3: Every process x_i with $i \neq 1$ receives $\langle z, 1\rangle$ during the first two L-tours before receiving $\langle L.id, 1\rangle$.

Proof of Claim 3: The first time x_{i-1} receives $\langle z, 0\rangle$ is before x_{i-1} receives $\langle L.id, 1\rangle$ in the first two L-tours, by Claims 1 and 2. In that step, x_{i-1} sends $\langle z, 1\rangle$. No process between x_{i-1} and x_i can stop that message. So, x_i receives $\langle z, 1\rangle$ during the first two L-tours before receiving $\langle L.id, 1\rangle$. ■

By Claims 2 and 3, each x_i receives the message $\langle z, 1\rangle$ during the first two L-tours before receiving $\langle L.id, 1\rangle$. Consider the first time x_i receives such a message. Then, $x_i.count = 1$. Either x_i is already passive and we are done, or $x_i.count$ is set to 2. Hence, when receiving $\langle L.id, 1\rangle$ during the first two L-tours, x_i executes **A5**-action and we are done. □

Lemma 6. *For any process p, if $p \neq L$, then p never executes* **A9**-*action.*

Proof. Assume, by the contradiction, that some process $p \neq L$ eventually executes **A9**-action. Let $x = p.id$. Then, p successively receives $\langle x, 0\rangle$, $\ldots$, $\langle x, k\rangle$ so that $p.active \wedge p.count = k$ holds when p receives $\langle x, k\rangle$. Notice also that p also receives $\langle L.id, 0\rangle$ and $\langle L.id, 1\rangle$, by Lemma 3.

First, p does not receive $\langle L.id, 0\rangle$ after $\langle x, k\rangle$, because otherwise p received at least $k+1$ messages tagged with label x during the first L-tour, which is impossible since the multiplicity of x is at most k and the links are FIFO.

Assume now that p receives $\langle L.id, 0\rangle$ before $\langle x, k\rangle$ but after $\langle x, 0\rangle$. Then, p is deactivated by **A5**-action when it receives $\langle L.id, 0\rangle$ because $p.count > 0$ and so before receiving $\langle x, k\rangle$, a contradiction.

So, p receives $\langle L.id, 0\rangle$ before $\langle x, 0\rangle$. Similarly, p does not receive $\langle L.id, 1\rangle$ after $\langle x, k\rangle$, because otherwise p received at least $k+1$ messages tagged with label x during the first L-tour. Then, p does not receive $\langle L.id, 1\rangle$ before $\langle x, 0\rangle$ because otherwise p does not receive any message tagged with x during the first L-tour, now it receives at least $\langle x, 0\rangle$ during the first L-tour from either its first predecessor with same label, or itself (if x is unique in the ring).

If p receives $\langle L.id, 1\rangle$ before $\langle x, 1\rangle$, then x is unique in the ring and when p receives $\langle L.id, 1\rangle$, p is deactivated by **A6**-action, and so before receiving $\langle x, k\rangle$, a contradiction.

Finally, if $k > 1$ and if p receives $\langle L.id, 1\rangle$ after $\langle x, 1\rangle$ but before $\langle x, k\rangle$, then p is deactivated by **A5**-action when it receives $\langle L.id, 1\rangle$, because $1 < p.count \leq k$. Hence, again, p is deactivated before receiving $\langle x, k\rangle$, a contradiction. □

Lemma 7. *In any execution of U_k:*

1. *For every process $p \neq L$, $p.active$ becomes* FALSE *within the first two L-tours.*
2. *For every process $p \neq L$, p never executes* **A9**-*action.*

3. *L executes* **A9***-action after exactly $k+1$ L-tours. In this action $L.leader \leftarrow L$, $L.isLeader \leftarrow$ TRUE, and $L.done \leftarrow$ TRUE.*
4. *For every process $p \neq L$ is a process, p executes* **A10***-action during the $(k+2)^{nd}$ L-tour. In this action $p.leader \leftarrow L$ and $p.done \leftarrow$ TRUE.*
5. *L executes* **A11***-action after exactly $k+2$ L-tours, and that is the last action of the execution.*

Proof. Part 1 follows from Lemmas 4 and 5. Part 2 is Lemma 6. Parts 3–5 follow from Corollary 5: The token initialized by L circles the ring $k+2$ times, each time incrementing $L.count$ once. At the end of the $(k+1)^{\text{st}}$ traversal, L executes **A9**-action, electing itself to be the leader. The message $\langle L.id, k+1\rangle$ then circles the ring, informing all other processes that L has been elected. Those latter processes halt after forwarding this message. When that final message reaches L, the execution is over. □

Theorem 5 below follows immediately from Lemma 7.

Theorem 5. *U_k solves the process-terminating leader election for $\mathcal{U}^* \cap \mathcal{K}_k$, for every given $k \geq 1$.*

Theorem 6. *U_k has time complexity at most $(k+2)n$, message complexity at most $3n^2 + (k-1)n$, and requires $\lceil \log(k+1) \rceil + 2b + 4$ bits in each process.*

Proof. Time complexity follows from Lemma 7.5. Space complexity follows from the definition of U_k. Consider now the message complexity of U_k. All tokens, except the one initiated by L, vanish during the three first L-tours, by Lemma 7.1. Consequently, only the token initiated by L circulates during the $k-1$ last L-tours. Hence, we have a message complexity of $3n^2+(k-1)n$ (at most $3n^2$ for messages transmitted during the 3 first L-tours, and exactly $(k-1)n$, with $k \leq n$, for the unique token circulating during the $k-1$ last L-tours). □

6. Algorithm A_k

We now give a solution, Algorithm A_k, to the process-terminating leader election for the class $\mathcal{A} \cap \mathcal{K}_k$, with fixed $k \geq 1$. A_k is based on the following observation. Consider a ring $\mathcal{R}$ of $\mathcal{A} \cap \mathcal{K}_k$ with n processes. As $\mathcal{R}$ is asymmetric, any two processes in $\mathcal{R}$ can be distinguished by examining all labels. So, using the lexicographical order, a process can be elected as the leader by examining all labels. Initially, any process p of $\mathcal{R}$ does not know the labels of $\mathcal{R}$, except its own. But, if each process broadcasts its own label clockwise, then any process can learn the labels of all other processes from messages it receives from its left neighbor. In the following, we show that, after examining finitely many labels, a process can decide that it learned (at least) all labels of $\mathcal{R}$ and so can determine whether it is the leader.

6.1. Variables of A_k

Each process p has six variables.

1. As defined in the specification, p has the constant $p.id$, the variables $p.leader$ (of label type), as well as $p.done$ and $p.isLeader$ (Booleans, initially FALSE).
2. Process p also has a Boolean variable $p.init$, initially TRUE.
3. Finally, p uses the variable $p.string$, a sequence of labels initially empty.

6.2. Messages of A_k

There are two kinds of messages: $\langle x \rangle$, where x is of label type, and $\langle$FINISH$\rangle$.

6.3. Overview of A_k

Sequences of Labels. Given any process p of $\mathcal{R}$, we define $LSeq(p)$, to be the infinite sequence of labels of processes, starting at p and continuing counterclockwise forever:

$$LSeq(p_i) = p_i.id, p_{i-1}.id, p_{i-2}.id \ldots, \text{ where subscripts are modulo } n.$$

For example, if the ring has three processes where $p_0.id = p_1.id = A$ and $p_2.id = B$, then $LSeq(p_0) = ABAABA\ldots$

For any sequence of labels σ, we define $\sigma_{|t}$ as the prefix of σ of length t, and $\sigma[i]$, for all $i \geq 1$, as the i^{th} element (starting from the left) of σ.

If σ is an infinite sequence (resp. a finite sequence of length λ), we say that $\pi = \sigma_{|m}$ is a *repeating prefix* of σ if $\sigma[i] = \pi[1 + (i-1) \bmod m]$ for all $i \geq 1$ (resp. for all $1 \leq i \leq \lambda$). Informally, if σ is infinite, then σ is the concatenation $\pi\pi\pi\ldots$ of infinitely many copies of π, otherwise σ is the truncation at length λ of the infinite sequence $\pi\pi\pi\ldots$

Let $srp(\sigma)$ be the *repeating prefix of σ of minimum length.* As $\mathcal{R}$ is asymmetric, we have:

Lemma 8. *Let p be a process and let $m \in \{2n, \ldots, \infty\}$. The length of $srp(LSeq(p)_{|m})$ is n.*

Proof. Let s be the smallest length of any repeating prefix of σ. $LSeq(p)_{|n}$ is a repeating prefix of σ and thus s is defined, and $s \leq n$.

If $s < n$, then the rotation by s is a non-trivial rotational symmetry of $\mathcal{R}$, contradicting the hypothesis that $\mathcal{R}$ is asymmetric. □

The next lemma shows that any process p can *fully determine* $\mathcal{R}$, *i.e.*, p can determine n, as well as the labeling of $\mathcal{R}$, from any prefix of $LSeq(p)$, provided that prefix contains at least $2k+1$ copies of any label.

Lemma 9. *Let p be a process, $m > 0$ and ℓ be a label. If $LSeq(p)_{|m}$ contains at least $2k+1$ copies of ℓ, then $\mathcal{R}$ is fully determined by $LSeq(p)_{|m}$.*

Proof. We note $\pi = LSeq(p)_{|m}$ and assume that it contains at least $2k+1$ copies of ℓ. First, $m > 2n$. Indeed, there are at most k copies of ℓ in any subsequence of $LSeq(p)$ of length no more than n, by definition of $\mathcal{K}_k$. So, at most $2k$ copies of ℓ in any subsequence of length no more than $2n$. Then, by Lemma 8, $srp(\pi) = LSeq(p)_{|n}$. Hence, one can compute $srp(\pi)$: its length provides n and its contents is exactly the counterclockwise sequence of labels in $\mathcal{R}$, starting from p. □

True Leader. We define the *true leader* of $\mathcal{R}$ as the process L such that $LSeq(L)_{|n}$ is a *Lyndon word* [26], *i.e.*, a non-empty string that is strictly smaller in lexicographic order than all of its rotations. In the following, we note $LW(\sigma)$ the rotation of the sequence σ which is a Lyndon word.

Algorithm 2: Actions of Process p in Algorithm A_k.

A1	::	$p.init$	$\to$	$p.string \leftarrow p.id$ $p.init \leftarrow$ FALSE **send** $\langle p.id \rangle$
A2	::	$\neg p.init \wedge$ **rcv** $\langle x \rangle \wedge \neg Leader(p.string \bullet x)$	$\to$	$p.string \leftarrow p.string \bullet x$ **send** $\langle x \rangle$
A3	::	$\neg p.init \wedge$ **rcv** $\langle x \rangle \wedge Leader(p.string \bullet x)$ $\wedge \neg p.isLeader$	$\to$	$p.string \leftarrow p.string \bullet x$ $p.isLeader \leftarrow$ TRUE $p.leader \leftarrow p.id$ $p.done \leftarrow$ TRUE **send** $\langle$FINISH$\rangle$
A4	::	$\neg p.init \wedge$ **rcv** $\langle$FINISH$\rangle \wedge \neg p.isLeader$	$\to$	$p.leader \leftarrow LW(srp(p.string))[1]$ $p.done \leftarrow$ TRUE **send** $\langle$FINISH$\rangle$ (halt)
A5	::	$\neg p.init \wedge$ **rcv** $\langle x \rangle \wedge p.isLeader$	$\to$	(nothing)
A6	::	$\neg p.init \wedge$ **rcv** $\langle$FINISH$\rangle \wedge p.isLeader$	$\to$	(halt)

In Algorithm A_k (see Algorithm 2), the true leader will be elected. Precisely, in A_k, a process p uses the variable $p.string$ to save a prefix of $LSeq(p)$ at any step: $p.string$ is initially empty and consists of all the labels that p has learnt during the execution of A_k so far. Lemma 9 shows how p can determine the label of the true leader. Indeed, if $p.string$ contains at least $2k+1$ copies of some label, $srp(p.string) = LSeq(p)_{|n}$. If $srp(p.string) = LW(srp(p.string))$, then p is the true leader. Otherwise, the label of the true leader is the first label of $LW(srp(p.string))$, *i.e.*, $LW(srp(p.string))[1]$.

In A_k, we use the function $Leader(\sigma)$ which returns TRUE if the sequence σ contains at least $2k+1$ copies of some label and $srp(\sigma) = LW(srp(\sigma))$, FALSE otherwise.

Phases of A_k. A_k consists of two phases, which we call the *string growth phase* and the *finishing phase*.

During the string growth phase, each process p builds a prefix of $LSeq(p)$ in $p.string$. First, p initiates a token containing its label, and also initializes $p.string$ to $p.id$ (**A1**-action). The token moves around the ring repeatedly until the end of the string growth phase. When p receives a label, p executes **A2**-action to append it to its string, and sends it to its right neighbor. Thus, each process keeps growing $p.string$.

Eventually, L receives a label x such that $L.string \bullet x$ is long enough for L to determine that it is the leader, see Lemma 9 and the definition of function $Leader$. In this case, L executes **A3**-action: L appends $L.string$ with x, ends the string growth phase, initiates the finishing phase by electing itself as leader, and sends the message $\langle$FINISH$\rangle$ to its right neighbor. The message $\langle$FINISH$\rangle$ traverses the ring, informing all processes that the election is over. As each process p receives the message (**A4**-action), it knows that a leader has been elected, can determine its label, $LW(srp(p.string))[1]$, and then halts. Meanwhile, L consumes every token (**A5**-action). When $\langle$FINISH$\rangle$ returns to L, it executes **A6**-action to halt, concluding the execution of A_k.

6.4. Correctness and Complexity Analysis

Theorem 7. *A_k solves the process-terminating leader election for $\mathcal{A} \cap \mathcal{K}_k$, for every given $k \geq 1$.*

Proof. Let $M = \max\{\text{mlty}(\ell) : \ell \text{ is a label in } \mathcal{R}\}$ and $m = \lceil (2k+1)/M \rceil n$. After receiving at most m messages containing labels (the messages cannot be discarded before the election of a leader, **A5**-action), by Lemma 9, every process will know $\mathcal{R}$ completely. Hence, by definition, L can determine that it is the true leader. As soon as L realizes that it is the leader, it will execute **A3**-action, sending the message $\langle$FINISH$\rangle$ around the ring.

Every process but L will receive the message $\langle$FINISH$\rangle$ and execute **A4**-action, which will be its final action. Finally L executes **A6**-action, ending the execution. So A_k solves the process-terminating leader election for $\mathcal{A} \cap \mathcal{K}_k$. □

Theorem 8. *A_k has time complexity at most $(2k+2)n$, has message complexity at most $(2k+1)n^2 + n$, and requires at most $(2k+1)nb + 2b + 3$ bits in each process.*

Proof. Let $M = \max\{\text{mlty}(\ell) : \ell \text{ is a label in } \mathcal{R}\}$ and $m = \lceil (2k+1)/M \rceil n$. After at most m time units, L can determine that it is the true leader and send a message $\langle$FINISH$\rangle$. In n additional time units, $\langle$FINISH$\rangle$ traverses the whole ring and comes back to L to conclude the execution. In the worst case, there are no duplicate labels, *i.e.*, $M = 1$. Hence, the time complexity of A_k is at most $(2k+2)n$ time units.

When the execution halts, all sent messages have been received. So, the number of message sendings is equal to the number of message receptions. Each token initiated at the beginning of the growing phase circulates in the ring until being consumed by L after it realizes that it is the true leader. Similarly, $\langle$FINISH$\rangle$ traverses the ring once and stopped at L. Hence, each process receives at most as many messages as L. L receives $2k+1$ messages with the same label x to detect that it is the true leader (**A3**-action). When L becomes leader, the received token $\langle x \rangle$ is consumed and L has received messages containing other labels (at most $n-1$ different labels) at most $2k$ times each. Then, L receives and consumes all other tokens (at most $n-1$) before receiving $\langle$FINISH$\rangle$. Overall, L receives at most $(2k+1)n+1$ messages and so, the message complexity is at most $(2k+1)n^2 + n$.

From the previous discussion, the length of $L.string$ is bounded by $2kn+1$. If $p \neq L$, then $p.string$ continues to grow after L executes **A3**-action until p executes **A4**-action by receiving the message $\langle$FINISH$\rangle$. Now, the FIFO property ensures that $p.string$ is appended at most $n-1$ times more than $L.string$ due to the remaining tokens. Thus the length of $p.string$ is always less than $(2k+1)n$. So, the space complexity is at most $(2k+1)nb+2b+3$ bits per process. □

7. Algorithm B_k

We now give another solution, Algorithm B_k, to the process-terminating leader election for the class $\mathcal{A} \cap \mathcal{K}_k$, with fixed $k \geq 1$. The space complexity of B_k is smaller than that of A_k, but its time complexity is greater. The state diagram and the actions of B_k are respectively given in Figure 6 and Algorithm 3.

Algorithm 3: Actions of Process p in Algorithm B_k.

A1	::	$p.state = \textsc{Init}$	$\rightarrow$	$p.state \leftarrow \textsc{Compute}$ $p.guest \leftarrow p.id$ **send** $\langle p.guest \rangle$
(Computation During a Phase)				
A2	::	$p.state = \textsc{Compute} \wedge \textbf{rcv}\ \langle x \rangle \wedge x > p.guest$	$\rightarrow$	(nothing)
A3	::	$p.state = \textsc{Compute} \wedge \textbf{rcv}\ \langle x \rangle \wedge x = p.guest$ $\wedge\ p.inner < k$	$\rightarrow$	$p.inner{+}{+}$ **send** $\langle x \rangle$
A4	::	$p.state = \textsc{Compute} \wedge \textbf{rcv}\ \langle x \rangle \wedge x < p.guest$	$\rightarrow$	$p.state \leftarrow \textsc{Passive}$ **send** $\langle x \rangle$
(Phase Switching)				
A5	::	$p.state = \textsc{Compute} \wedge \textbf{rcv}\ \langle x \rangle \wedge x = p.guest$ $\wedge\ p.inner = k$	$\rightarrow$	$p.state \leftarrow \textsc{Shift}$ **send** $\langle \textsc{Phase\_Shift}, p.guest \rangle$
A6	::	$p.state = \textsc{Shift} \wedge \textbf{rcv}\ \langle \textsc{Phase\_Shift}, x \rangle$ $\wedge\ (x \neq p.id \vee p.outer < k)$	$\rightarrow$	$p.state \leftarrow \textsc{Compute}$ **if** $p.id = x$ **then** $p.outer{+}{+}$ $p.guest \leftarrow x$ $p.inner \leftarrow 1$ **send** $\langle p.guest \rangle$
(Passive Processes)				
A7	::	$p.state = \textsc{Passive} \wedge \textbf{rcv}\ \langle x \rangle$	$\rightarrow$	**send** $\langle x \rangle$
A8	::	$p.state = \textsc{Passive} \wedge \textbf{rcv}\ \langle \textsc{Phase\_Shift}, x \rangle$	$\rightarrow$	**send** $\langle \textsc{Phase\_Shift}, p.guest \rangle$ $p.guest \leftarrow x$
(Ending Phase)				
A9	::	$p.state = \textsc{Shift} \wedge \textbf{rcv}\ \langle \textsc{Phase\_Shift}, x \rangle$ $\wedge\ x = p.id \wedge p.outer = k$	$\rightarrow$	$p.state \leftarrow \textsc{Win}$ $p.isLeader \leftarrow \textsc{True}$ $p.leader \leftarrow p.id$ $p.guest \leftarrow p.id$ **send** $\langle \textsc{Finish}, p.id \rangle$
A10	::	$p.state = \textsc{Passive} \wedge \textbf{rcv}\ \langle \textsc{Finish}, x \rangle$	$\rightarrow$	$p.state \leftarrow \textsc{Halt}$ $p.leader \leftarrow x$ $p.done \leftarrow \textsc{True}$ **send** $\langle \textsc{Finish}, x \rangle$ (halt)
A11	::	$p.state = \textsc{Win} \wedge \textbf{rcv}\ \langle \textsc{Finish}, x \rangle$	$\rightarrow$	$p.state \leftarrow \textsc{Halt}$ $p.done \leftarrow \textsc{True}$ (halt)

7.1. Variables of B_k

Each process p has seven variables.

1. As defined in the specification, p has the constant $p.id$, the variables $p.leader$ (of label type), as well as $p.done$ and $p.isLeader$ (Booleans, initially FALSE).
2. Process p also maintains a variable $p.state \in \{$INIT, COMPUTE, SHIFT, PASSIVE, WIN, HALT$\}$, initially equals to INIT. If $p.state$ = PASSIVE, then p is said to be *passive*, otherwise p is said to be *active*. Notice also that HALT is the terminal state for every process.
3. Finally, p has two counters $p.inner$ and $p.outer$ of range $1 \ldots k$, both initially equal to 1.

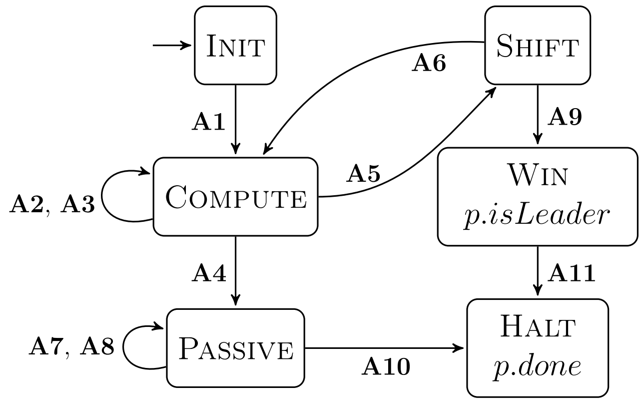


Figure 6: State diagram of B_k.

7.2. Messages of B_k

B_k executes by phases. Three kinds of message are exchanged to manage these phases: the token $\langle x \rangle$ is used during the computation of a phase, $\langle \text{PHASE\_SHIFT}, x \rangle$ is used to notify that a phase is over, and $\langle \text{FINISH}, x \rangle$ is used during the ending phase, where x is of label type. Intuitively, we say that a process is in its i^{th} phase, with $i \geq 1$, if it received $i-1$ messages $\langle \text{PHASE\_SHIFT}, \_ \rangle$.

7.3. Overview of B_k

Let $k \geq 1$ and $\mathcal{R} \in \mathcal{A} \cap \mathcal{K}_k$. Like A_k, B_k elects the true leader of $\mathcal{R}$, namely, the process $L \in \mathcal{R}$ such that $LSeq(L)_{|n}$ is a Lyndon word, *i.e.*, $LSeq(L)_{|n}$ is minimum among the sequences $LSeq(q)_{|n}$ of all processes $q \in \mathcal{R}$, where sequences are compared using lexicographical ordering.

During the execution, the processes that are (still) competing to be the leader are the active ones. Initially, the set of active processes contains all processes: $Act_0 = \{p_0, ..., p_{n-1}\}$. An execution consists of phases where processes are *deactivated*, *i.e.*, become passive. At the end of a given phase $i \geq 1$, the set of active processes is given by:

$$Act_i = \left\{ p \in \mathcal{R} : LSeq(p)_{|i} = LSeq(L)_{|i} \right\}$$

see Figure 7. During phase $i \geq 1$, a process q is removed from Act_i, when $LSeq(q)[i] > LSeq(L)[i]$; more precisely, when q realizes that some process $p \in Act_{i-1}$ satisfies $LSeq(p)[i] < LSeq(q)[i]$. When $i \geq n$, Act_i is reduced to $\{L\}$, since $\mathcal{R}$ is asymmetric. Using k, B_k is able to detect that at least n phases have been done, and so to terminate.

Phase Computation. The goal of the i^{th} phase is to compute Act_i, given Act_{i-1}, namely to deactivate each active process p such that $LSeq(p)[i] > LSeq(L)[i]$. To that purpose, p assigns $p.guest$ in such way that $p.guest = LSeq(p)[i]$ during phase i. (How $p.guest$ is maintained in each phase will be explained later.)

During phase $i \geq 1$, the value $p.guest$ of every active process p circulates among active processes: at the beginning of the phase, every active process sends its current *guest* to its

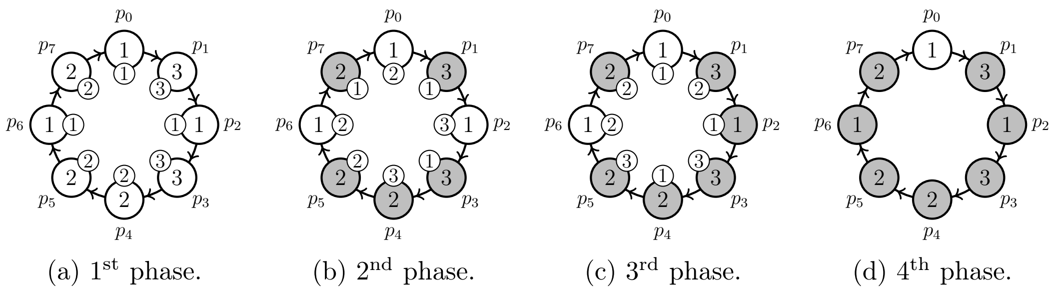


Figure 7: Extracts from an example of execution of B_k where $k = 3$, showing the active (in white) and passive (in gray) processes at the beginning of each phase. The *guest* of a process is in the white bubble next to the corresponding node.

right neighbor (**A1**-action for the first phase, **A6**-action for other phases). Since passive processes are no more candidate, they simply forward the token (**A7**-action). When an active process p receives a label x greater than $p.guest$, it discards this value (**A2**-action), since $x > p.guest \geq LSeq(L)[i]$. Conversely, when p is active and receives a label x lower than $p.guest$, it turns to be passive, executing **A4**-action; nevertheless, it forwards x.

A process p, which is (still) active, can end the computation of its phase i once it has considered the *guest* value of every other process that are active all along phase i (*i.e.*, processes in Act_{i-1} that did not become passive during phase i). Such a process p detects the end of the current phase when it has seen its current guest value (*i.e.*, $p.guest$) $k+1$ times. To that goal, we use the counter variable $p.inner$, which is initialized to 1 at the beginning of each phase ($p.inner$ is initialized to 1 and reset at each **A6**-action) and incremented each time p receives the value $p.guest$ while being active (**A3**-action) (once a process is passive the variable *inner* is meaningless). So, the current phase ends for an active process p when it receives $p.guest$ while $p.inner$ was already equal to k (**A5**-action).

Phase Switching. We now explain how $p.guest$ is maintained at each phase. Initially, $p.guest$ is set to $p.id$ and phase 1 starts for p (**A1**-action). Next, the value of $p.guest$ for every p is updated when switching to the next phase.

First, note that it is mandatory that every (active and passive) process updates its *guest* variable when entering a new phase, *i.e.*, after detecting the end of the previous phase, so that the labels that circulate during the computation of the phase actually represent $LSeq(p)[i]$ for process $p \in Act_{i-1}$. Now, FIFO links allow to enforce a barrier synchronization as follows.

At the end of phase $i \geq 1$, Act_i is computed, and every still active process p has the same label prefix of length i, $LSeq(p)_{|i}$, hence the same value for $p.guest = LSeq(p)[i]$. They are all able to detect the end of phase i. So, they switch their *state* from COMPUTE to SHIFT and signal the end of the phase by sending a token $\langle \text{PHASE\_SHIFT}, p.guest \rangle$ (**A5**-action).

Tokens $\langle \text{PHASE\_SHIFT}, \_ \rangle$ circulate in the ring, through passive processes (**A8**-action) until reaching another (or possibly the same) active process: when a process p (being passive or active) receives $\langle \text{PHASE\_SHIFT}, x \rangle$:

1. it switches from phase i to $i + 1$ by adopting x as new *guest* value, and

2. if p is passive, it sends $\langle$PHASE_SHIFT$, y\rangle$ where y was its previous *guest* value; otherwise, the shifting process is done and so p switches *p.state* from SHIFT to COMPUTE or WIN and starts a new phase (**A6**-action or **A9**-action).

As a result, all *guest* values have eventually shifted by one process on the right for the next phase.

Due to FIFO links and the fact that active processes switch to state SHIFT between two successive phases, phases cannot overlap, *i.e.*, when a label x is considered in phase i by any active process in state COMPUTE, x is the *guest* of some process q which is active in phase i, such that $LSeq(q)[i] = x$.

Number of Phases. Phase switching stops for an active process p once *p.guest* has shifted $k+1$ times to its own label *p.id*. Indeed, when *p.guest* is shifted to *p.id* for the $(k+1)^{\text{th}}$ times, it is guaranteed that the number of phases executed by the algorithm is greater or equal to n, because $p.guest = LSeq(p)[i]$ in phase i and there is no more than k processes with the label *p.id*. In this case, p is the true leader and every other process q is passive.

To detect this, we use at each process p the counter *p.outer*. It is initialized to 1 and incremented by each active process p at each phase switching when the new guest value is equal to *p.id* (**A6**-action). When *p.outer* reaches the value $k+1$ (or equivalently when p receives $\langle$PHASE_SHIFT$, p.id\rangle$ while $p.outer = k$, see **A9**-action), p declares itself as the leader and initiates the final phase: it sends a token $\langle$FINISH$, p.id\rangle$; each other process successively receives the token (**A10**-action), saves the label in the token in its *leader* variable, forwards the token, and then halts. Once the token reaches p again (**A11**-action), it also halts.

7.4. Correctness and Complexity Analysis

Throughout this section, we consider an arbitrary ring $\mathcal{R}$ of $\mathcal{A} \cap \mathcal{K}_k$, with fixed $k \geq 1$. To prove the correctness of B_k (Theorem 9), we first establish that its phases are well-defined (see Lemma 10), *e.g.*, they do not overlap. Then, Lemmas 11-16 prove the invariant of the algorithm, by induction on the phase number. Finally, Theorem 10 gives a complexity analysis of B_k. (All the proofs below are easier to follow using the state diagram of Figure 6.)

A process p *is in Phase* $i \geq 0$ if it shifts i times the value of its variable *p.guest*: the shift precisely occurs when p executes the assignment of *p.guest*. Such shifts occur in **A1**-action, **A6**-action, **A9**-action, or **A8**-action. Notice that the three later actions are executed upon the reception of some token $\langle$PHASE_SHIFT$, _\rangle$. As we will see below, a barrier synchronization is achieved between each phase using these tokens.

Lemma 10. *Let $i \geq 1$. A message received in Phase i has been sent in Phase i. Conversely, if a message has been sent in Phase i, it can only be received in Phase i.*

Proof. First, we prove some preliminary results.

Claim 1: Between two shifts at *p.guest*, each process p has sent and received at least one message.

Proof of Claim 1: A process p can only update *p.guest* by executing **A1**, **A6**, **A8**, or **A9**-action. Furthermore, **A1**-action is always executed first and only once. Assume p updates

$p.guest$ in $\gamma_i \mapsto \gamma_{i+1}$ and later in $\gamma_j \mapsto \gamma_{j+1}$. So, p executes **A6**, **A8**, or **A9**-action in $\gamma_j \mapsto \gamma_{j+1}$. Now, if p executes **A8**-action, it receives a message $\langle$PHASE_SHIFT, _$\rangle$ and sends $\langle$PHASE_SHIFT, $p.guest\rangle$ before updating $p.guest$ during $\gamma_j \mapsto \gamma_{j+1}$. If p executes **A6** or **A9**-action, it receives a message $\langle$PHASE_SHIFT, _$\rangle$ in step $\gamma_j \mapsto \gamma_{j+1}$ before updating $p.guest$. Moreover, in that case, the previous action p has executed is necessarily **A5**-action, so p sent a message $\langle$PHASE_SHIFT, $p.guest\rangle$ between γ_{i+1} and γ_j. ∎

Assume now, by the contradiction, that some process q receives in Phase $j \geq 0$ a message m sent by its predecessor p in Phase $i \geq 0$ such that $i \neq j$. Without loss of generality, assume m is the first message subject to that condition.

Claim 2: $i \geq 1$ and $j \geq 1$

Proof of Claim 2: p cannot send messages before executing **A1**-action, *i.e.*, before setting $p.guest$ to $p.id$ and starting its first phase. So, $i \geq 1$. Similarly, q cannot receive any message before executing **A1**-action, so $j \geq 1$. ∎

Now, since m is the first problematic message, by Claim 1 and owing the fact that the links are FIFO, we can deduce that $j = i - 1$ or $j = i + 1$. Let consider the two cases.

- If $j = i + 1$, then q updates its *guest* once more than p. Let consider the last time q updates its *guest* before receiving m, *i.e.*, the last time q executes **A1**, **A6**, **A8**, or **A9**-action to switch from its $(j-1)^{\text{th}}$ to its j^{th} phase (*i.e.*, to switch from its i^{th} to its j^{th} phase). By Claim 2, $i \geq 1$, so $j \geq 2$, and so, it does no execute **A1**-action to switch from its $(j-1)^{\text{th}}$ to its j^{th} phase, since **A1** is always executed first and only once.
  Now, if q executes **A6**, **A8**, or **A9**-action, it receives a message m' of the form $\langle$PHASE_SHIFT, _$\rangle$ in Phase $j-1$. Since m is the first problematic message, m' was sent by p in Phase $j-1$. Either p executes **A8**-action and switches to Phase j, or p executes **A5**-action to send m'. Now, in this latter case, p necessarily executes **A6**-action to send the next message after m', and p switches to Phase j before sending that message. In both cases, p switches to Phase j before sending m, a contradiction.
- If $j = i - 1$, then p updates its *guest* once more than q. Let consider the last time p updates its *guest* before sending m, *i.e.*, the last time p executes **A1**, **A6**, **A8**, or **A9**-action to switch from its $(i-1)^{\text{th}}$ to its i^{th} phase. Let consider each cases:
  - If p executes **A1**-action, then $i = 1$ since **A1** is always executed first and only once. Now, by Claim 2, $j \geq 1$, a contradiction.
  - If p executes **A6** or **A9**-action, it necessarily executes **A5**-action beforehand and so sends a message $m' = \langle$PHASE_SHIFT, _$\rangle$ to q in Phase $i-1$. Since m is the first problematic message, q receives m' in Phase $i-1$ executing **A6**, **A8**, or **A9**-action. Either action makes q switching from Phase $i-1$ to Phase i before receiving m, a contradiction.
  - If p executes **A8**-action, it sends a message $m' = \langle$PHASE_SHIFT, _$\rangle$ before switching to phase i. Again, since m is the first problematic message, q receives m' in Phase $i-1$ executing **A6**, **A8**, or **A9**-action. Either action makes q switching from Phase $i-1$ to Phase i before receiving m, a contradiction.

□

In the following, we say that a process p_i is *deadlocked* if p_i is disabled although a message m is ready to be received by p_i, *i.e.*, p_i is disabled while m is the head message of $S_{(p_{i-1},p_i)}$.

Definition 3 (HI_i)**.** *Let* $X = \min\left\{x : LSeq(L)_{|x} \text{ contains } k+1 \text{ occurrences of } L.id\right\}$*. For any* $i \in \{1, \ldots, X\}$*, we define* HI_i *as the following predicate:* $\forall p \in \mathcal{R}, \forall j, 1 \leq j < i$,

1. *p.guest is equal to* $LSeq(p)[j]$ *in Phase* j*,*
2. p *is not deadlocked during Phase* j*, and*
3. p *eventually exits Phase* j *and,* $p \in Act_j$ *if and only if* p *exits its phase* j *using* **A6** *or* **A9**-*action.*

Lemma 11. *For all* $i \in \{1, \ldots, X\}$*,* HI_i *holds.*

Lemma 11 is proven by induction on i. The base case ($i = 1$) is trivial. The induction step (assume HI_i and show HI_{i+1}, for $i \in \{1, \ldots, X-1\}$) consists in proving the correct behavior of Phase i. To that goal, we prove Lemmas 12, 15, and 16 which respectively show Conditions 1, 2, and 3 for HI_{i+1}.

Lemma 12. *For* $i \in \{1, \ldots, X-1\}$*, if* HI_i *holds, then* $\forall p \in \mathcal{R}, \forall j < i+1$*, p.guest is equal to* $LSeq(p)[j]$ *in Phase* j*.*

Proof. Let $i \in \{1, \ldots, X-1\}$ such that HI_i holds. First note that for every process p, we have $LSeq(p)[1] = p.id = p.guest$ in Phase 1. Hence the lemma holds for $i = 1$. Now assume that $i > 1$. By HI_i, we have that for every $1 \leq j < i$, $LSeq(p)[j] = p.guest$ at Phase j.

We consider now the case of Phase i. Since $i > 1$, a process can only update its variable *guest* using **A6**, **A8** or **A9**-action, namely during phase switching (**A1**-action is always executed first and only once). Let p be a process at Phase i (by 3, any process eventually enters Phase i) and consider, in the execution, the step where p switches from Phase $i-1$ to Phase i: p receives from its left neighbor, q, a message $\langle \text{PHASE\_SHIFT}, x\rangle$, where x was the value of $q.guest$ when q sent the message (see **A5** and **A8**-actions). From Lemma 10, and since p receives it at Phase $i-1$, q sends this message at Phase $i-1$ also. Hence, $x = q.guest$ at Phase $i-1$. Now, when p receives the message, it assigns its variable $p.guest$ to x (**A6**, **A8**, or **A9**-action): hence, at Phase i, $p.guest = LSeq(q)[i-1] = LSeq(p)[i]$. $\square$

By Lemma 10, if p receives $\langle \text{PHASE\_SHIFT}, \_\rangle$ at Phase $i \geq 1$, it was sent by its left neighbor in Phase i. So, by Lemma 12, we have:

Corollary 6. *For* $i \in \{1, \ldots, X-1\}$*, if* HI_i *holds, then* $\forall p \in \mathcal{R}$*, if* p *exits Phase* $j \leq i$ *by* **A9**-*action, then* $LSeq(p)[j]$ *equals p.id.*

Lemma 13. *For* $i \in \{1, \ldots, X-1\}$*, if* HI_i *holds, then no* **A9**-*action is executed before Phase* $i+1$*.*

Proof. Assume by the contradiction that HI_i holds and some **A9**-action is executed before Phase $i+1$. Consider the first time it occurs: assume some process p executes **A9**-action in some Phase $j \leq i$. By Corollary 6, using **A9**-action, p receives a message $\langle \text{PHASE\_SHIFT}, x\rangle$

with $x = p.id = LSeq(p)[j]$. Furthermore, we have that $p.outer = k$ at Phase j. Hence $p.id$ was observed $k+1$ times since the beginning of the execution: $p.guest$ shifted k times to the value $p.id$ and the value x in the received message is also $p.id$. By Lemma 12, the sequence of values of $p.guest$ is equal to $LSeq(p)_{|j-1}$. Adding $x = LSeq(p)[j]$ at the end of the sequence, we obtain $LSeq(p)_{|j}$. Hence, $j = \min\{x : LSeq(p)_{|x}$ contains $p.id$ $(k+1)$ times$\}$ and $n < j$ (this implies that $j \geq 2$, hence $j-1 \geq 1$). As p executes **A9**-action in Phase j, it is active during its whole j^{th} phase and hence exits its phase $j-1$ using **A6**-action. By Condition 3 in HI_i and since $j-1 < i$, $p \in Act_{j-1}$. By definition of Act_{j-1}, since $j > n$, $Act_{j-1} = \{L\}$, hence $p = L$. As a consequence, $j = X$, a contradiction. □

In the following, we show that processes cannot deadlock (Lemma 15).

Lemma 14. *While a process is in state* Compute *(resp.* Shift*), the next message it has to consider cannot be of the form* $\langle$Phase_Shift, _$\rangle$ *(resp.* $\langle x \rangle$*).*

Proof. Assume by the contradiction that some process p is in state Compute (resp. Shift), but receives an unexpected message $\langle$Phase_Shift, _$\rangle$ (resp. $\langle x \rangle$) meanwhile. We only examine the first case, the other case being similar. The unexpected corresponding token may have been transmitted through passive processes to p, but was first initiated by some active process q (**A5**-action). Since **A5**-action was enabled at process q, q received k messages $\langle q.guest \rangle$ during one and the same phase. By the multiplicity, for at least one of those messages, the corresponding token m was initiated by q using **A1** or **A6**-action. So, m has traversed the entire ring using **A3**-**A5**, and **A7**-actions. Lemma 10 ensures that this traversal occurs during one and the same phase. As a consequence, $q.guest \leq r.guest$ for every process r that were active when receiving m (none of the aforementioned actions provoke a phase shifting). In particular, $q.guest \leq p.guest$.

As q executed **A5**-action, k messages $\langle q.guest \rangle$ were sent by q (one action, either **A1** or **A6**-action, and $k-1$ **A3**-actions) during the traversal of m, and so during the same phase again. Hence, p has also received $\langle q.guest \rangle$ k times during the same phase. Thus, $p.guest \leq q.guest$ since p is still active, and so $p.guest = q.guest$. Now, counters *inner* of p and q counted accordingly during this phase: $p.inner$ should be greater than or equal to k. Hence p should have executed **A5**-action before receiving the unexpected message, a contradiction. □

Lemma 15. *For every* $i \in \{1, \ldots, X-1\}$*, if* HI_i *holds, then* $\forall p \in \mathcal{R}$*,* p *is not deadlocked before Phase* $i+1$*.*

Proof. Let $i \in \{1, \ldots, X-1\}$ such that HI_i holds. Let p be any process. If p is in state Init or Passive in Phase i, then it cannot deadlock since the states Init and Passive are not blocking by definition of the algorithm. From Lemma 13 since HI_i holds, p cannot take state Win before Phase $i+1$. Hence, it cannot take state Halt by **A11**-action. As no **A9**-action is executed during Phase i, no message $\langle$Finish, _$\rangle$ circulates in the ring during this phase (Lemma 10): **A10**-action cannot be enabled, hence p cannot take state Halt by **A10**-action as well. If p is in state Compute (resp. Shift), it cannot receive any message $\langle$Phase_Shift, _$\rangle$ (resp. $\langle x \rangle$) by Lemma 14. Moreover, it cannot have received any message $\langle$FINISH, _$\rangle$ since no such message was sent during this phase (see Lemma 13 which applies as HI_i holds). As a conclusion, there is no way for p to deadlock during Phase i. □

Lemma 16. *For every $i \in \{1, \ldots, X-1\}$, if HI_i holds, then $\forall p \in \mathcal{R}$, $\forall j < i+1$, p eventually exits Phase j and, $p \in Act_j$ if and only if p exits its phase j by* **A6** *or* **A9***-action.*

Proof. Let $i \in \{1, \ldots, X-1\}$ such that HI_i holds.

Claim 1: $\forall p$, if $p \in Act_{i-1}$ (resp. $\notin Act_{i-1}$), p initiates (resp. does not initiate) a token $\langle LSeq(p)[i]\rangle$ (resp. any token) at the beginning of Phase i.

Proof of Claim 1: If $i = 1$, every process p is in Act_0 and starts its phase 1, *i.e.*, its execution, by executing **A1**-action and sending its label $p.id = LSeq(p)[1]$. Otherwise ($i > 1$), by Lemma 13, no process can execute **A9**-action before Phase $i+1$. So by HI_i, every process $p \in Act_{i-1}$ exits Phase $i-1$, and so starts Phase i, by executing **A6**-action and sending its label $p.guest = LSeq(p)[i]$ (Lemma 12). By HI_i, if p is not in Act_{i-1}, p cannot exit Phase $i-1$ by executing **A6**-action and so it cannot initiate a token $\langle p.id\rangle$ at the beginning of Phase i. ■

Claim 2: Any process p receives $\langle LSeq(L)[i]\rangle$ k times during its phase i.

Proof of Claim 2: Consider a token $m = \langle LSeq(L)[i]\rangle$ that circulates the ring (at least one is circulating since $L \in Act_{i-1}$ initiates one at the beginning of Phase i, see Claim 1). m is always received in Phase i (see Lemma 10) all along its ring traversal. From HI_i and Lemma 15, no process is deadlocked before its phase $i+1$. Hence, when m reaches a process in state PASSIVE, it is forwarded (**A7**-action) and when m reaches a process q in state COMPUTE (with $q.guest = LSeq(q)[i] \geq LSeq(L)[i]$, by Lemma 12 and definition of L), it is also forwarded unless **A5**-action is enabled at q. Eventually, **A5**-action occurs at every process q (at least one, *e.g.*, L) such that $LSeq(q)[i] = LSeq(L)[i]$, since $q.inner$ is initialized to 1 at the beginning of the phase (**A1** or **A6**-action) and incremented if q receives $LSeq(q)[i]$. Hence, q has received k messages $\langle LSeq(L)[i]\rangle$ during the phase.

As a consequence, between any two processes q and q' in Act_{i-1} (in state COMPUTE in Phase i, see HI_i) such that $LSeq(q)[i] = LSeq(q')[i] = LSeq(L)[i]$, k tokens $\langle LSeq(L)[i]\rangle$ circulates during phase i; any process between q and q' has forwarded them (and so received them). ■

By HI_i, the lemma holds for all $j < i$. Let now consider the case $j = i$. If $p \in Act_i$, then $LSeq(p)_{|i} = LSeq(L)_{|i}$ and in particular, $LSeq(p)[i] = LSeq(L)[i]$. As $Act_i \subseteq Act_{i-1}$, p is active at the end of Phase $i-1$ and as no **A9**-action can take place before Phase $i+1$ (Lemma 13), p is in state COMPUTE during the computation of Phase i. Since $p.guest = LSeq(L)[i] \leq LSeq(q)[i]$ for any $q \in Act_{i-1}$ (Lemma 12, definition of L), and as any token $\langle x\rangle$ that circulates during the Phase is initiated by some process $q \in Act_{i-1}$ with $x = LSeq(q)[i]$ (HI_i and Claim 1), p never executes **A4**-action during Phase i. Furthermore, p receives k times $p.guest$ during the phase (Claim 2), hence it executes **A5**-action followed by **A6** or **A9**-action to exit Phase i.

Conversely, if $p \notin Act_i$, it may be or not in Act_{i-1}. If $p \notin Act_{i-1}$, then from HI_i, p exits Phase $i-1$ by **A8**-action; it remains in state PASSIVE all along Phase i and can only exit Phase i by **A8**-action. Otherwise, $p \in Act_{i-1}$, *i.e.*, $LSeq(p)_{|i-1} = LSeq(L)_{|i-1}$ but $LSeq(p)[i] > LSeq(L)[i]$. p executes **A4**-action at least when receiving the first occurrence of $\langle LSeq(L)[i] \rangle$ (Claim 2) and takes state PASSIVE. Once p is passive, it remains so and can only exit Phase i using **A8**-action.

Finally, at least L eventually executes **A5**-action: the phase switching occurs (started by L or another process) causing all processes to exit Phase i. □

This ends the proof of Lemma 11.

Theorem 9. *B_k solves the process-terminating leader election for $\mathcal{A} \cap \mathcal{K}_k$.*

Proof. By Lemma 11 and definition of X, Phase X eventually starts and L is the only process that exits Phase X executing **A6** or **A9**-action. Now, by Lemma 11 and Corollary 6, $\forall i \in \{1, \ldots, X\}$, $L.guest = LSeq(L)[i]$ during Phase i. Hence, when L begins its X^{th} phase, it is the $(k+1)^{\text{th}}$ time that it sets $L.guest$ to $L.id$. Since $L.outer$ is initialized to 1 and incremented when L enters a new phase with $L.guest = L.id$, L enters its phase X by **A9**-action. So, L sends a token $\langle \text{FINISH}, L.id \rangle$. L also sets $L.isLeader$ and $L.leader$ to TRUE and $L.id$, resp. Every other process p receives the token in Phase X (Lemma 10) while being in state PASSIVE, since p exits its $(X-1)^{\text{th}}$ phase executing **A8**-action (Lemma 11). So, p saves $L.id$ in its variable *leader*, then transmits the token to its right neighbor, and finally halts (**A10**-action). Eventually L receives the token and halts (**A11**-action). □

Theorem 10. *B_k has time complexity at most $(k+1)^2n^2$, message complexity at most $2k^2n^2 + (3k+1)n^2 + (1-2k)n$, and requires $2\lceil \log k \rceil + 3b + 5$ bits per process.*

Proof. A phase ends when an active process sees its *guest* $k+1$ times. This requires $(k+1)n$ time units. There is exactly X phases and $X \leq (k+1)n$. Thus, the time complexity of B_k is at most $(k+1)^2n^2$.

During the first phase, every process starts by sending its *id*. Since a phase involves at most $(k+1)n$ actions per process, each process forwards labels at most $(k+1)n$ times. Finally, to end the first phase, every process sends and receives $\langle \text{PHASE\_SHIFT}, \_ \rangle$. Hence, at most $(k+1)n^2 + n$ messages are sent during the first phase. Moreover, only processes that have the same label as L (at most k) are still active after the first phase.

For every phase $i > 1$, let $d = \text{mlty}(\min\{p.guest : p \in Act_{i-1}\})$. When phase i starts, every active process (at most k) sends its new *guest*. When the first message ends its first traversal (kn messages), every process that becomes passive in the phase is already passive. Then, the variables *inner* of the remaining active processes increment of d each tour of ring by a message. So the remaining messages (at most d) do at most $\frac{k}{d}$ traversals (n hops): at most kn messages. Overall, the phase requires at most $2kn$ messages exchanged. As there is at most $(k+1)n - 1$ phases after the first one, overall there are at most $2k^2n^2 + (3k+1)n^2 + (1-2k)n$ messages exchanged.

Finally, for every process p, $p.inner$ and $p.outer$ are initialized to 1 and they are never incremented over than k. Hence, every process requires $2\lceil \log k \rceil + 3b + 5$ bits. □

Class	Impossibility results
$\mathcal{K}_k$	Message-terminating leader election impossible
$\mathcal{U}^*$	Process-terminating leader election impossible
$\mathcal{A}$	Process-terminating leader election impossible

Class	**Time** (process-terminating)	**Bits exchanged** (message-terminating)
$\mathcal{U}^* \cap \mathcal{K}_k$	$\Omega(kn)$	$\Omega(kn + n^2)$
$\mathcal{A} \cap \mathcal{K}_k$	$\Omega(kn)$	$\Omega(kn + n^2)$

Class	Algorithm	Time	Messages	Memory
$\mathcal{U}^* \cap \mathcal{K}_k$	U_k	$\leq (k+2)n$	$\leq 3n^2 + (k-1)n$	$\lceil \log(k+1) \rceil + 2b + 4$
$\mathcal{A} \cap \mathcal{K}_k$	A_k	$\leq (2k+2)n$	$\leq (2k+1)n^2 + n$	$2(k+1)nb + 2b + 3$
	B_k	$\leq (k+1)^2 n^2$	$\leq 2k^2n^2 + (3k+1)n^2 + (1-2k)n$	$2\lceil \log k \rceil + 3b + 5$

Table 1: Summary of the results.

8. Conclusion and Perspectives

We have studied the leader election problem in unidirectional ring networks with homonym processes. Our results are summarized in Table 1. Our process-terminating algorithm for $\mathcal{U}^* \cap \mathcal{K}_k$ (with $k \geq 1$) is asymptotically optimal in both time and memory requirements. We have also proposed two process-terminating algorithms, A_k and B_k, for the more general class $\mathcal{A} \cap \mathcal{K}_k$ (with $k \geq 1$). A_k is asymptotically optimal in time ($O(kn)$), but requires an important memory requirement ($O(knb)$ bits per process). In contrast, B_k is asymptotically optimal in terms of memory requirement ($O(\log k + b)$), but has a higher time complexity ($O(k^2n^2)$).

8.1. Short-term Perspectives

Finding the best trade-off process-terminating algorithm for the class $\mathcal{A} \cap \mathcal{K}_k$ is a direct extension of our work. Furthermore, the amount of bits exchanged in an execution of U_k ($O((kn + n^2)b)$ bits, where b is the number of bits required to store a label) is very close to the lower bound we have proven ($\Omega(kn + n^2)$ bits). (*N.b.*, it is typically assumed that $b = O(\log n)$.) On the contrary, the number of bits exchanged in an execution of A_k or B_k (respectively $O(n^2(2k+1)b)$ and $O(k^2n^2b)$) is greater. Whether or not it is possible to reduce the amount of exchanged information without degrading the other performances of the algorithms is worth investigating.

8.2. Long-term Perspectives

An interesting long-term perspective of our work would be to investigate fault-tolerant leader election in homonymous systems.

Crash Failures. In the context of *crash failures*, leader election in fully-identified rings have already been investigated in [27]: after a crash, the topology is re-organized as a ring where the node with the highest calculated priority is elected. Notice that this paper also considers the join of new nodes. Then, to the best of our knowledge, in homonymous crash-prone systems, only the *consensus* problem has been studied in fully-connected networks; see [28]. However, consensus is easier to solve than leader election in message passing since leader election requires a perfect failure detector, while consensus can be achieved with an unreliable one [29]. Hence, finding the weakest assumptions under which leader election is solvable in homonymous crash-prone systems is very challenging.

Byzantine Failures. If we address now *Byzantine failures* (which model malicious process behaviors) in the context of homonymous message passing systems, only results about the consensus problem (also called *Byzantine agreement* in that context) are available [30, 31]. Again, consensus is easier to solve than leader election in these settings. Indeed, a perfect Byzantine detector is necessary to solve leader election, while consensus can be performed using an unreliable one [32]. Again, this makes very attractive the problem achieving leader election in homonymous Byzantine-prone systems under weak assumptions.

Rational Processes. Finally, the basic aim of Byzantine processes is to make the election fail, *e.g.*, by electing two processes. Another interesting approach is to consider a coalition of so-called *rational processes* that try to influence the result of the election, *e.g.*, to elect one member of their group. The problem of election that withstands such collusions has been introduced and investigated as the *fair leader election* problem in a recent paper [33]. In particular, authors propose a randomized solution for ring networks which assumes that the processes know the whole set of identifiers. Weakening such an assumption while additionally considering homonymous processes is also a very challenging perspective of our work.

References

[1] D. Angluin, Local and global properties in networks of processors, in: Proceedings of STOC'80, 1980, pp. 82–93 (1980).

[2] Z. Xu, P. K. Srimani, Self-stabilizing anonymous leader election in a tree, Intl. J. of Foundations of Computer Science 17 (2) (2006) 323–336 (2006).

[3] N. A. Lynch, Distributed Algorithms, Morgan Kaufmann Publishers Inc., 1996 (1996).

[4] I. Stoica, R. T. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, H. Balakrishnan, Chord: a scalable peer-to-peer lookup protocol for internet applications, IEEE/ACM Trans. Netw. 11 (1) (2003) 17–32 (2003). doi:10.1109/TNET.2002.808407.
URL `https://doi.org/10.1109/TNET.2002.808407`

[5] A. I. T. Rowstron, P. Druschel, Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems, in: R. Guerraoui (Ed.), Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12-16, 2001, Proceedings, Vol. 2218 of Lecture Notes in Computer Science, Springer, 2001, pp. 329–350 (2001). doi:10.1007/3-540-45518-3_18.
URL `https://doi.org/10.1007/3-540-45518-3_18`

[6] P. Boldi, S. Vigna, An effective characterization of computability in anonymous networks, in: J. L. Welch (Ed.), Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings, Vol. 2180 of Lecture Notes in Computer Science, Springer, 2001, pp. 33–47 (2001). doi:10.1007/3-540-45414-4_3.
URL https://doi.org/10.1007/3-540-45414-4_3

[7] H. Attiya, A. Gorbach, S. Moran, Computing in totally anonymous asynchronous shared memory systems, Inf. Comput. 173 (2) (2002) 162–183 (2002). doi:10.1006/inco.2001.3119.
URL https://doi.org/10.1006/inco.2001.3119

[8] H. Buhrman, A. Panconesi, R. Silvestri, P. M. B. Vitányi, On the importance of having an identity or, is consensus really universal?, Distributed Comput. 18 (3) (2006) 167–176 (2006). doi:10.1007/s00446-005-0121-z.
URL https://doi.org/10.1007/s00446-005-0121-z

[9] D. Chaum, E. van Heyst, Group signatures, in: EUROCRYPT, 1991, pp. 257–265 (1991).

[10] R. L. Rivest, A. Shamir, Y. Tauman, How to leak a secret, in: C. Boyd (Ed.), Advances in Cryptology — ASIACRYPT 2001, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, pp. 552–565 (2001).

[11] H. Pan, E. Hou, N. Ansari, Re-note: An e-voting scheme based on ring signature and clash attack protection, in: 2013 IEEE Global Communications Conference (GLOBECOM), 2013, pp. 867–871 (2013).

[12] P. P. Tsang, V. K. Wei, Short linkable ring signatures for e-voting, e-cash and attestation, in: R. H. Deng, F. Bao, H. Pang, J. Zhou (Eds.), Information Security Practice and Experience, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 48–60 (2005).

[13] D. Mödinger, H. Kopp, F. Kargl, F. J. Hauck, A flexible network approach to privacy of blockchain transactions, in: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), 2018, pp. 1486–1491 (2018).

[14] A. Shaker, D. Reeves, Self-stabilizing structured ring topology p2p systems, in: Proceedings - Fifth IEEE International Conference on Peer-to-Peer Computing, P2P 2005, Vol. 2005, 2005, pp. 39– 46 (01 2005). doi:10.1109/P2P.2005.34.

[15] A. Forestiero, C. Mastroianni, Description of the self-chord P2P application, in: G. Fortino, A. Garro, L. Palopoli, W. Russo, G. Spezzano (Eds.), Proceedings of the 12th Workshop on Objects and Agents, Rende (CS), Italy, Jul 4-6, 2011, Vol. 741 of CEUR Workshop Proceedings, CEUR-WS.org, 2011, pp. 175–177 (2011).
URL http://ceur-ws.org/Vol-741/DEM03_ForestieroMastroianni.pdf

[16] M. Yamashita, T. Kameda, Electing a leader when processor identity numbers are not distinct (extended abstract), in: Proceedings of WDAG'89, 1989, pp. 303–314 (1989).

[17] P. Flocchini, E. Kranakis, D. Krizanc, F. L. Luccio, N. Santoro, Sorting and election in anonymous asynchronous rings, Journal of Parallel and Distributed Computing 64 (2) (2004) 254–265 (2004).

[18] S. Dobrev, A. Pelc, Leader election in rings with nonunique labels, Fundamenta Informaticae 59 (4) (2004) 333–347 (2004).

[19] J. Chalopin, E. Godard, Y. Métivier, Election in partially anonymous networks with arbitrary knowledge in message passing systems, Distributed Computing 25 (4) (2012) 297–311 (2012).

[20] C. Delporte-Gallet, H. Fauconnier, H. Tran-The, Leader election in rings with homonyms, in: Proceedings of NETYS'14, 2014, pp. 9–24 (2014).

[21] D. Dereniowski, A. Pelc, Topology recognition and leader election in colored networks, Theoretical Computer Science 621 (2016) 92–102 (2016).

[22] K. Altisen, A. K. Datta, S. Devismes, A. Durand, L. L. Larmore, Leader election in rings with bounded multiplicity (short paper), in: Proceedings of SSS'16, 2016, pp. 1–6 (2016).

[23] K. Altisen, A. K. Datta, S. Devismes, A. Durand, L. L. Larmore, Leader election in asymmetric labeled unidirectional rings, in: Proceedings of IPDPS'17, 2017, pp. 182–191 (2017).

[24] G. Tel, Introduction to Distributed Algorithms, Cambridge Univ. Press, 2000 (2000).

[25] G. L. Peterson, An o(n log n) unidirectional algorithm for the circular extrema problem, ACM Transactions on Programming Languages and Systems 4 (4) (1982) 758–762 (1982).

[26] R. C. Lyndon, On burnside's problem, Transactions of the American Mathematical Society 77 (1954) 202–215 (1954).

[27] T. Biswas, R. Bhardwaj, A. Ray, P. Kuila, A novel leader election algorithm based on resources for ring networks, International Journal of Communication Systems 31 (2018) e3583 (04 2018). doi:10.1002/dac.3583.

[28] C. Delporte-Gallet, H. Fauconnier, H. Tran-The, Uniform consensus with homonyms and omission failures, in: D. Frey, M. Raynal, S. Sarkar, R. K. Shyamasundar, P. Sinha (Eds.), Distributed Computing and Networking, 14th International Conference, ICDCN 2013, Mumbai, India, January 3-6, 2013. Proceedings, Vol. 7730 of Lecture Notes in Computer Science, Springer, 2013, pp. 161–175 (2013). doi:10.1007/978-3-642-35668-1_12.
URL https://doi.org/10.1007/978-3-642-35668-1_12

[29] S.-H. Park, About the relationship between election problem and failure detector in asynchronous distributed systems, in: Proceedings of the 1st International Conference on Computational Science: PartI, ICCS'03, Springer-Verlag, Berlin, Heidelberg, 2003, p. 185–193 (2003).

[30] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, A. Kermarrec, E. Ruppert, H. Tran-The, Byzantine agreement with homonyms, Dist. Comp. 26 (5-6) (2013) 321–340 (2013).

[31] C. Delporte-Gallet, H. Fauconnier, H. Tran-The, Byzantine agreement with homonyms in synchronous systems, Theor. Comput. Sci. 496 (2013) 34–49 (2013). doi:10.1016/j.tcs.2012.11.012.
URL `https://doi.org/10.1016/j.tcs.2012.11.012`

[32] K. P. Kihlstrom, L. E. Moser, P. M. Melliar-Smith, Byzantine fault detectors for solving consensus, The Computer Journal 46 (1) (2003) 16–35 (2003).

[33] A. Yifrach, Y. Mansour, Fair leader election for rational agents in asynchronous rings and networks, in: C. Newport, I. Keidar (Eds.), Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018, ACM, 2018, pp. 217–226 (2018). doi:10.1145/3212734.3212767.
URL `https://doi.org/10.1145/3212734.3212767`