Leader Election in Asymmetric Labeled Unidirectional Rings

Karine Altisen,* Ajoy K. Datta,† Stéphane Devismes,* Anaïs Durand,* and Lawrence L. Larmore†
*Université Grenoble Alpes, Grenoble, France
†UNLV, Las Vegas, USA

***Abstract*—We study (deterministic) leader election in unidirectional rings of homonym processes that have no *a priori* knowledge on the number of processes. In this context, we show that there is no algorithm that solves process-terminating leader election for the class of asymmetric labeled rings. In particular, there is no process-terminating leader election algorithm in rings in which at least one label is unique. However, we show that process-terminating leader election is possible for the subclass of asymmetric rings, where multiplicity is bounded. We confirm this positive results by proposing two algorithms, which achieve the classical trade-off between time and space.**

***Keywords*-Leader Election, Homonym Processes, Multiplicity, Unidirectional Rings.**

I. Introduction

In 1980, Angluin [1] showed the impossibility of solving deterministic leader election in networks of anonymous processes. This negative result led to two major opposite lines of research. The first one consists in circumventing 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].

Quite recently, the notion of *homonym processes* [4], [5] 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 mainly motivated by the *group signatures* [6]: signatures are labels, and process groups share the same signature to maintain some kind of privacy. Homonyms have been mainly studied for solving the consensus problem in networks where processes are subjected to Byzantine failures [5], [7].

Related Work: Several works address deterministic solutions for various election problems in *rings of homonym processes*, *e.g.*, [4], [8]–[10]. In [8], Flocchini *et al* consider the *weak leader election* problem in *bidirectional* rings of homonym processes. The weak leader election problem consists in electing at least one, but at most two processes, and in the latter case the two processes should be neighbors. Assuming that processes *a priori* know the number of processes n, they show that *process-terminating* (*i.e.*, every process eventually halts) weak leader election is possible if and only if the ring is asymmetrically labeled, *i.e.*, looking at the sequence of all labels in any sense of direction, there is no non-trivial rotational symmetry. They also propose two process-terminating algorithms for asymmetric labeled rings of n processes, assuming that n is prime and only two labels are possible, 0 or 1. The first algorithm additionally assumes a common sense of direction; while the second one is a generalization of the first one, where the common sense of direction is removed.

Delporte *et al* [9] have investigated the leader election problem in *bidirectional rings* of homonym processes. They have given a necessary and sufficient condition on the number of distinct labels needed to design a leader election algorithm. Precisely, they show that there exists a deterministic solution for *message-terminating* (*i.e.*, processes do not halt but only a finite number of messages are exchanged) leader election on a bidirectional ring if and only if the number of labels is strictly greater than the greatest proper divisor of n. Assuming this condition, they give two algorithms. The first one is message-terminating and does not assume any further extra knowledge. The second one assumes the processes know n, is process-terminating, and is asymptotically optimal in messages ($O(n \log n)$).

In [4], Dobrev and Pelc consider a generalization of the process-terminating leader election in both bidirectional and unidirectional rings of homonym processes which *a priori* know 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 give synchronous algorithms for bidirectional and unidirectional rings working in time $O(M)$ using $O(n \log n)$ messages. They also give an asynchronous algorithm

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

for bidirectional rings that uses $O(nM)$ messages, and show that it is optimal; no time complexity is given.

In [10], we focus on *unidirectional* rings of homonym processes, where processes know neither the number of processes n, nor any bound on it. We propose a process-terminating algorithm, assuming that (1) at least one process holds a label which is unique, and (2) processes *a priori* know an upper bound k on the *multiplicity* of the labels, *i.e.*, no label occurs more than k times. The proposed algorithm has time complexity at most $(k+2)n$ and message complexity $O(n^2+kn)$.

Contribution: In this work, we explore the design of *process-terminating* (deterministic) leader election algorithms in unidirectional rings of homonym processes which, contrary to [4], [8], [9], know neither the number of processes n, nor any bound on it. Since we only consider *unidirectional* rings, results from Delporte *et al* [9] do not apply: the common sense of direction may help processes to solve the leader election problem.

We first consider the class $\mathcal{U}^*$ of all unidirectional rings in which at least one label is unique. We show that without further knowledge, there is no process-terminating leader election algorithm for this class. (Recall that if we additionally assume the knowledge of some upper bound on the multiplicity, the problem becomes solvable [10].)

We then consider the class $\mathcal{U}^* \cap \mathcal{K}_k$, where $\mathcal{K}_k$ is the class of all ring networks whose multiplicity is less than or equal to k; in this class, processes know k *a priori*. We show the lower bound $\Omega(kn)$ on the time complexity of any leader election in $\mathcal{U}^* \cap \mathcal{K}_k$. This result implies that the algorithm for $\mathcal{U}^* \cap \mathcal{K}_k$, proposed in [10], is asymptotically optimal in time.

Finally, we consider more general settings: the class $\mathcal{A}$ of all asymmetric labeled rings, *i.e.*, all labeled ring networks that have no non-trivial rotational symmetry. First, since $\mathcal{U}^* \subseteq \mathcal{A}$, there is also no process-terminating leader election for the class $\mathcal{A}$. Hence, we consider the subclass $\mathcal{A} \cap \mathcal{K}_k$. We propose two process-terminating leader election algorithms for this class which achieve the classical trade-off between time and space. Based on the lower bound for $\mathcal{U}^* \cap \mathcal{K}_k$ and the fact that $\mathcal{U}^* \cap \mathcal{K}_k \subseteq \mathcal{A} \cap \mathcal{K}_k$, the first one is asymptotically optimal in time, as its time complexity is at most $(2k+2)n$. Moreover, its message and space complexity are at most $n^2(2k+1)$ messages and $O(knb)$ bits per process, respectively (b is the number of bits required to store any label). In the second one, we reduce the space complexity to $O(\lceil \log k \rceil + b)$ bits, but at the price of increasing both time and message complexities to $O(k^2n^2)$.

Finally, notice that, perhaps surprisingly, there are labeled rings (*e.g.*, a ring of three processes with labels 1, 2, and 2) for which we can solve process-terminating leader election, whereas it cannot be solved in the model of [4], [9]. This suggests that the knowledge of k and of a common orientation is more helpful to solve process-terminating leader election in a ring than the knowledge of n or bounds on n.

Roadmap: The model and definitions are proposed in Section II. Impossibility results and lower bounds are exposed in Section III. Our algorithms, their correctness and complexity analysis are given in Sections IV-V.

II. Preliminaries

Ring Networks: We assume unidirectional rings of $n \geq 2$ processes, p_0, ..., 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.

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 $S_{(p_i,p_{i+1})}$. We 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 R, let *mlty*$[\ell]$, the *multiplicity* of ℓ in R, be the number of processes in 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 $\langle guard \rangle \rightarrow \langle statement \rangle$. 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 assume that the local algorithm of each process p contains at most one action triggerable without the reception of any message. This action is 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* [11] 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: An algorithm ALG solves the *process-terminating leader election* problem (leader election for short) in a ring network R if every execution e of ALG on R satisfies the following conditions: e is finite and for every process p,

1) p has a Boolean variable $p.isLeader$ which is initially FALSE, never switches from TRUE to FALSE (the decision of being the leader is irrevocable), and, in the terminal configuration of e, is TRUE for a unique process L, the leader. So, there should be at most one leader in each configuration.
2) p has a variable $p.leader$ such that in the terminal configuration of e, $p.leader = L.id$.
3) p has a Boolean variable $p.done$, initially FALSE, such that $p.done$ is TRUE in the terminal configuration of e, indicating that p knows that the leader has been elected. Once $p.done$ becomes TRUE, it will never again become FALSE, $L.isLeader$ is equal to TRUE, and $p.leader$ is permanently set to $L.id$.
4) p eventually *halts* (local termination decision) after $p.done$ becomes TRUE.

Ring Network Classes: An algorithm ALG solves the leader election *for the class of ring network* $\mathcal{R}$ if ALG solves the leader election for every network $R \in \mathcal{R}$. It is important to note that, for ALG to be a leader election algorithm for a class $\mathcal{R}$, ALG cannot be given any specific information about the network (such as its cardinality or the actual multiplicity of the ring) unless that information holds for all members of $\mathcal{R}$, since ALG must work for every $R \in \mathcal{R}$ without any change whatsoever in its code.

We say that a ring network R is *symmetric* if it has a non-trivial rotational symmetry, *i.e.*, there is some integer $0 < d < n$ such that p_{i+d} and p_i have the same label for all i. Otherwise, we say R is *asymmetric*.

We now define three important classes of ring networks. $\mathcal{K}_k$ is the class of all ring networks such that no label occurs more than k times, where $k \geq 1$ is a given integer. $\mathcal{A}$ is the class of all asymmetric ring networks. $\mathcal{U}^*$ is the class of all rings in which at least one label is unique. By definition, $\mathcal{U}^* \subseteq \mathcal{A}$.

III. Impossibility Results and Lower bounds

Recall that an execution is *synchronous* if at each step all enabled processes execute.

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

Proof: Let $k \geq 2$ and ALG be a leader election algorithm for $\mathcal{U}^* \cap \mathcal{K}_k$. Let $R_n \in \mathcal{K}_1$ be a ring of n processes, noted $p_0, \ldots, p_{n-1}$ with distinct labels $l_0, \ldots, l_{n-1}$ respectively. Since $\mathcal{K}_1 \subseteq \mathcal{U}^* \cap \mathcal{K}_k$, ALG is correct for R_n and so, the synchronous execution $\Gamma = \gamma_0, \ldots$ of ALG on R_n is finite. Let T be the number of steps of Γ: within T synchronous steps, $p_\ell.isLeader$ becomes TRUE for some $0 \leq \ell \leq n-1$, *i.e.,* p_ℓ is the leader in the terminal configuration γ_T of Γ.

We now build the ring $R_{n,k} \in \mathcal{U}^* \cap \mathcal{K}_k$ of $kn + 1$ processes, $q_0, \ldots, q_{kn}$, with labels consisting of the sequence $l_0, \ldots, l_{n-1}$ repeated k times, followed by a single label $X \notin \{l_0, \ldots, l_{n-1}\}$. Since $R_{n,k} \in \mathcal{U}^* \cap$

$\mathcal{K}_k$, ALG is correct on $R_{n,k}$. Let $\Gamma' = \gamma'_0, \ldots$ be the synchronous execution of ALG on $R_{n,k}$.

By construction, we have the following property on Γ': (*) 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. Indeed, no information from q_{kn} has already reached q_j.

Assume, by the contradiction, that $T \leq (k-2)n$. Let $j_1 = (k-2)n + \ell$ and $j_2 = (k-1)n + \ell$. Since $\ell \in \{0, ..., n-1\}$, we have $j_1, j_2 \in \{(k-2)n, ..., kn-1\}$, *i.e.,* $T \leq j_1 < j_2 < kn$. Then, $j_1 \bmod n = j_2 \bmod n = \ell$. 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, $q_{j_1}.isLeader = q_{j_2}.isLeader =$ TRUE in γ'_T. This contradicts the fact that ALG achieves leader election in $R_{n,k}$. (Bullet 1 of the specification is violated in γ'_T, see page 3.) Hence, the number of steps 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 2. *Let $k \geq 2$. The time complexity of any algorithm that solves the leader election for $\mathcal{U}^* \cap \mathcal{K}_k$ is $\Omega(k\,n)$, where n is the number of processes.*

Theorem 1. *There is no algorithm that solves the leader election for $\mathcal{U}^*$.*

Proof: Suppose ALG is an algorithm for $\mathcal{U}^*$. Let R_n be a ring network of $\mathcal{K}_1$ with n processes. Let e be the synchronous execution of ALG on R_n: as $\mathcal{K}_1 \subseteq \mathcal{U}^*$, ALG is correct for 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 1 and Corollary 2 imply the following two corollaries.

Corollary 3. *There is no algorithm that solves the leader election for $\mathcal{A}$.*

Corollary 4. *Let $k \geq 2$. The time complexity of any algorithm that solves the leader election for $\mathcal{A} \cap \mathcal{K}_k$ is $\Omega(k\,n)$, where n is the number of processes.*

IV. ALGORITHM A_k

We now give a solution, Algorithm A_k, to the leader election for the class $\mathcal{A} \cap \mathcal{K}_k$, for fixed k. A_k is based on the following observation. Consider a ring R of $\mathcal{A} \cap \mathcal{K}_k$ with n processes. As R is asymmetric, any two processes in 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 R does not know the labels of 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 learnt (at least) all labels of R and so can determine whether it is the leader.

Sequences of Labels: Given any process p of R, we define *LLabels*(p), to be the infinite sequence of labels of processes, starting at p and continuing counter-clockwise forever: *LLabels*$(p_i) = p_i.id, p_{i-1}.id, p_{i-2}.id \ldots$, 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 *LLabels*$(p_0) = ABAABA\ldots$ For any sequence of labels σ, we define σ^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*(σ) be the *repeating prefix of σ of minimum length.*

As R is asymmetric, we have:

Lemma 5. *Let p be a process and $m \in \{2n, \ldots, \infty\}$. The length of srp(LLabels$(p)^m$) is n.*

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

Lemma 6. *Let p be a process, $m > 0$ and ℓ be a label. If LLabels$(p)^m$ contains at least $2k+1$ copies of ℓ, then R is fully determined by LLabels$(p)^m$.*

Proof: We note $\pi =$ *LLabels*$(p)^m$ 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 *LLabels*(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 5, *srp*$(\pi) =$ *LLabels*$(p)^n$. Hence, one can compute *srp*(π): its length provides n and its contents is exactly the counter-clockwise sequence of labels in R, starting from p. ■

True Leader: We define the *true leader* of R as the process L such that *LLabels*$(L)^n$ is a *Lyndon word* [12], *i.e.,* a non-empty string that is strictly smaller in lexicographic order than all of its rotations. In the following, we note *LW*(σ) the rotation of the sequence

Table 1: Actions of Process p in Algorithm A_k

A1 p.INIT	$\rightarrow$	p.INIT $\leftarrow$ FALSE, *p.string* $\leftarrow$ *p.id*, **send**$\langle$ *p.id* $\rangle$
A2 $\neg p$.INIT $\wedge$ **rcv**$\langle x \rangle \wedge \neg$*Leader*(*p.string* . x)	$\rightarrow$	*p.string* $\leftarrow$ *p.string* . x, **send**$\langle x \rangle$
A3 $\neg p$.INIT $\wedge$ **rcv**$\langle x \rangle \wedge$ *Leader*(*p.string* . x) $\wedge \neg$*p.isLeader*	$\rightarrow$	*p.string* $\leftarrow$ *p.string* . 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*	$\rightarrow$	*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*	$\rightarrow$	(nothing)
A6 $\neg p$.INIT $\wedge$ **rcv**$\langle$ FINISH $\rangle \wedge$ *p.isLeader*	$\rightarrow$	(halt)

σ which is a Lyndon word.

In Algorithm A_k (see Table 1), the true leader will be elected. Precisely, in A_k, a process p uses a variable *p.string* to save a prefix of *LLabels*(p) at any step: *p.string* is initially empty and consists of all the labels that p has received during the execution of A_k so far. Lemma 6 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*) = *LLabels*(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*(σ) which returns TRUE if the sequence σ contains at least $2k+1$ copies of some label and *srp*(σ) = *LW*(*srp*(σ)), FALSE otherwise.

Overview of A_k*:* Each process p has six variables. As defined in the specification, p has the variables *p.id* and *p.leader* (of label type), and *p.done* and *p.isLeader* (Booleans, initially FALSE). p also has a Boolean variable p.INIT, initially TRUE, and the variable *p.string*, as defined above. There are two kinds of messages: $\langle x \rangle$ where x is of label type and $\langle$FINISH$\rangle$.

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 *LLabels*(p) in *p.string*. First, p initiates a token containing its label, and also initializes *p.string* to *p.id* (Action A1). The token moves around the ring repeatedly until the end of the string growth phase. When p receives a label, p executes Action A2 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* . x is long enough for L to determine that it is the leader, see Lemma 6 and the definition of *Leader*. In this case, L executes Action A3: 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 (Action A4), 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 (Action A5). When $\langle$FINISH$\rangle$ returns to L, it executes Action A6 and halts, concluding the execution of A_k.

Theorem 2. A_k *solves the leader election for* $\mathcal{A} \cap \mathcal{K}_k$*, has time complexity at most* $(2k+2)n$*, message complexity at most* $n^2(2k+1)$*, and requires at most* $(2k+1)nb + 2b + 3$ *bits in each process.*

Proof: Let $M = \max\{mlty[\,\ell\,] : \ell \text{ is a label in } R\}$ and $m = \lceil (2k+1)/M \rceil\, n$. After at most m time units, by Lemma 6, every process will know R completely, hence, by definition, L can determine that it is the the true leader. As soon as L realizes that it is the leader, it will execute Action A3, sending the message $\langle$FINISH$\rangle$ around the ring. Every process but L will receive the message $\langle$FINISH$\rangle$ and execute Action A4, which will be its final action, within at most $n-1$ time units. Finally L executes Action A6 at most one time unit later, ending the execution. Thus, every process p halts after fewer than $m+n$ time units. In the worst case, there are no duplicate labels, *i.e.,* $M = 1$, so A_k solves the leader election for $\mathcal{A} \cap \mathcal{K}_k$ and its time complexity is at most $(2k+2)n$.

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 (Action A3). 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 $n(2k+1)+1$ messages and so, the message complexity is at most $n^2(2k+1)+n$.

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 Action A3 until p executes Action

A4 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. ■

V. ALGORITHM B_k

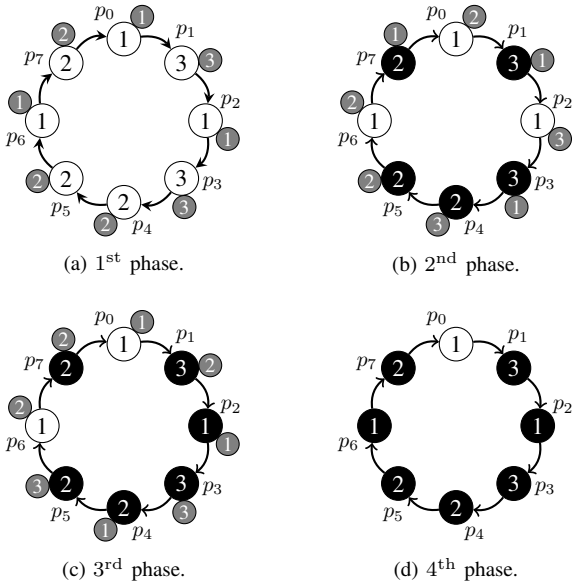
For any $k \geq 2$, we now give another leader election algorithm, B_k, for a ring R in the class $\mathcal{A} \cap \mathcal{K}_k$. The space complexity of B_k is smaller than that of A_k, but its time complexity is greater. See Table 2 for its code and Figure 2 for its state diagram.

Overview: Like A_k, B_k elects the true leader of R, namely, the process L such that $LLabels(L)^n$ is a Lyndon word, *i.e.*, $LLabels(L)^n$ is minimum among the sequences $LLabels(q)^n$ of all processes q, where sequences are compared using lexicographical ordering.

We define as *active* the processes that are (still) competing to be the leader (other processes are said to be *passive*). We compute the lexicographical ordering step by step, as follows. Initially, the set of active processes contains all processes: $Act_0 = \{p_0, ..., p_{n-1}\}$. An execution of B_k 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: $Act_i = \{p \in R, LLabels(p)^i = LLabels(L)^i\}$. During phase $i \geq 1$, a process q is removed from Act_i, when $LLabels(q)[i] > LLabels(L)[i]$; more precisely, when q realizes that some process $p \in Act_{i-1}$ satisfies $LLabels(p)[i] < LLabels(q)[i]$, see Figure 1. When $i \geq n$, Act_i is reduced to $\{L\}$, since R is asymmetric. Using k, B_k is able to detect that at least n phases have been done, and so to terminate.

As defined in the specification, we use at each process p the constant $p.id$ and the variables $p.leader$ (of label type), $p.done$ and $p.isLeader$ (Booleans, initially FALSE). Each process p also maintains a variable $p.state \in \{$INIT, COMPUTE, SHIFT, PASSIVE, WIN, HALT$\}$, initially equals to INIT. A passive process (*i.e.*, no more competing) is in state PASSIVE; other states are used by (still) active processes; state HALT is the last state for every process. Three kinds of message are exchanged: $\langle x \rangle$ is used during the computation of a phase, $\langle$PHASE_SHIFT$, x\rangle$ is used to notify that a phase is over, and $\langle$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 $(i-1)$ $\langle$PHASE_SHIFT$, _\rangle$ messages.

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 $LLabels(p)[i] > LLabels(L)[i]$. To that purpose, we introduce, at each process p, a variable $p.guest$, of label type, such that $p.guest = LLabels(p)[i]$. (How $p.guest$ is maintained in each phase will be explained later.)



(a) 1^{st} phase. (b) 2^{nd} phase.

(c) 3^{rd} phase. (d) 4^{th} phase.

Figure 1: Execution of B_k where $k = 3$ and p_0 is elected. We color in white (resp. in black) processes that are active (resp. passive) at the beginning of the phase. The gray label next to a process shows its $guest$.

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 right neighbor (Action B1 for the first phase, Action B6 for other phases). Since passive processes are no more candidate, they simply forward the message (Action B7). When an active process p receives a label x greater than $p.guest$, it discards this value (Action B2), since $x > p.guest \geq LLabels(L)[i]$. Conversely, when p is active and receives a label x lower than $p.guest$, it turns to be passive, executing Action B4 (nevertheless, p 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 the value $p.guest$ $(k+1)$ times. To that goal, we use the counter variable $p.inner$, which is initialized to 1 at the beginning of each phase (Actions B1 and B6) and incremented each time p receives the value $p.guest$ while being active (Action B3) (once a process is passive the variable *inner*

Table 2: Actions of Process p in Algorithm B_k

B1	$p.state = \text{INIT}$	$\rightarrow$	$p.state \leftarrow \text{COMPUTE}$, $p.guest \leftarrow p.id$ $p.inner \leftarrow 1, p.outer \leftarrow 1$, **send**$\langle p.guest \rangle$
Computation during a phase			
B2	$p.state = \text{COMPUTE} \wedge$ **rcv**$\langle x \rangle \wedge x > p.guest$	$\rightarrow$	(nothing)
B3	$p.state = \text{COMPUTE} \wedge$ **rcv**$\langle x \rangle \wedge x = p.guest \wedge p.inner < k$	$\rightarrow$	$p.inner$ ++, **send**$\langle x \rangle$
B4	$p.state = \text{COMPUTE} \wedge$ **rcv**$\langle x \rangle \wedge x < p.guest$	$\rightarrow$	$p.state \leftarrow \text{PASSIVE}$, **send**$\langle x \rangle$
Phase Switching			
B5	$p.state = \text{COMPUTE} \wedge$ **rcv**$\langle x \rangle \wedge x = p.guest \wedge p.inner = k$	$\rightarrow$	$p.state \leftarrow \text{SHIFT}$, **send**$\langle \text{PHASE\_SHIFT}, p.guest \rangle$
B6	$p.state = \text{SHIFT} \wedge$ **rcv**$\langle \text{PHASE\_SHIFT}, x \rangle \wedge (x \neq p.id \vee p.outer < k)$	$\rightarrow$	$p.state \leftarrow \text{COMPUTE}$, **if** $p.id = x$ **then** $p.outer$ ++ $p.guest \leftarrow x$, $p.inner \leftarrow 1$, **send**$\langle p.guest \rangle$
Passive Processes			
B7	$p.state = \text{PASSIVE} \wedge$ **rcv**$\langle x \rangle$	$\rightarrow$	**send**$\langle x \rangle$
B8	$p.state = \text{PASSIVE} \wedge$ **rcv**$\langle \text{PHASE\_SHIFT}, x \rangle$	$\rightarrow$	**send**$\langle \text{PHASE\_SHIFT}, p.guest \rangle$, $p.guest \leftarrow x$
Ending Phase			
B9	$p.state = \text{SHIFT} \wedge$ **rcv**$\langle \text{PHASE\_SHIFT}, x \rangle \wedge x = p.id \wedge p.outer = k$	$\rightarrow$	$p.state \leftarrow \text{WIN}$, $p.isLeader \leftarrow \text{TRUE}$ $p.leader \leftarrow p.id$, $p.guest \leftarrow p.id$, **send**$\langle \text{FINISH}, p.id \rangle$
B10	$p.state = \text{PASSIVE} \wedge$ **rcv**$\langle \text{FINISH}, x \rangle$	$\rightarrow$	$p.state \leftarrow \text{HALT}$, **send**$\langle \text{FINISH}, x \rangle$ $p.leader \leftarrow x$, $p.done \leftarrow \text{TRUE}$, $(halt)$
B11	$p.state = \text{WIN} \wedge$ **rcv**$\langle \text{FINISH}, x \rangle$	$\rightarrow$	$p.state \leftarrow \text{HALT}$, $p.done \leftarrow \text{TRUE}$, $(halt)$

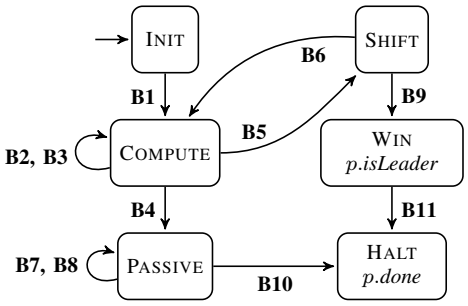


Figure 2: State diagram of B_k.

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 (Action B5).

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 (Action B1). 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 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 $LLabels(p)[i]$ 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 has the same label prefix of length i, $LLabels(p)^i$, hence the same value for $p.guest = LLabels(p)[i]$. As a consequence, 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 message $\langle \text{PHASE\_SHIFT}, p.guest \rangle$ (Action B5). Messages $\langle \text{PHASE\_SHIFT}, \_ \rangle$ circulate in the ring, through passive processes (Action B8) 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 $\langle \text{PHASE\_SHIFT}, y \rangle$ 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 (Action B6 or B9). As a result, all $guest$ values have eventually shifted by one on the right for the next phase.

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

How Many Phases?: Phase switching stops for an active process p once its $guest$ took the value $p.id$ $(k+1)$ times. Indeed, when $p.guest$ is updated for the $(k+1)^{\text{th}}$ times by $p.id$, it is guaranteed that the number of phases executed by the algorithm is greater or equal to n, because $p.guest = LLabels(p)[i]$ and there is no more than k processes with the same value $p.id$. In this case, p is the true leader and every other process q is passive. Again, to detect this, we use at each process p a counter called $p.outer$. It is initially set to 1 (Action B1) and incremented by each active

process at each phase switching (Action B6). When $p.outer$ reaches the value $k+1$ (or equivalently when p receives $p.id$ while $p.outer = k$, see Action B9), p declares itself as the leader and initiates the final phase: it sends a message $\langle \text{FINISH}, p.id \rangle$; each other process successively receives the message, saves the label in the message in its *leader* variable, forwards the message, and then halts. Once the message reaches the leader (p) again, it also halts.

Correctness and Complexity of B_k*:* To prove the correctness of B_k (Theorem 3), we first establish that phases are causally well-defined (see Observation 1), *e.g.* they do not overlap. Then, lemmas 7-13 prove the invariant of the algorithm, by induction on the phase number. Finally, Theorem 4 proves its complexity.

A barrier synchronization is achieved between each phase using messages $\langle \text{PHASE\_SHIFT}, \_ \rangle$. Hence we have the following observation:

Observation 1. *Let $i \geq 1$. A message received in phase i has been sent in phase i (it was actually initiated in phase i). Conversely, if a message has been sent in phase i, it can only be received in phase i.*

In the following, we say that a process p is *deadlocked* if p is disabled although a message is ready to be received by p. We let $X = \min\{x : \mathit{LLabels}(L)^x \text{ contains } L.id \ (k+1) \text{ times}\}$. For any $i \in \{1, \ldots, X\}$, we define HI_i as the following predicate: $\forall p \in R, \forall j, 1 \leq j < i$,

1) $p.guest$ is equal to $\mathit{LLabels}(p)[j]$ in phase j,
2) p is not deadlocked during its phase j, and
3) $p \in Act_j$ if and only if p exits its phase j using Action B6 or B9.

Lemma 7. *For all $i \in \{1, \ldots, X\}$, HI_i holds.*

Lemma 7 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 8, 12, and 13 which respectively show Conditions 1, 2, and 3 for HI_{i+1}.

Lemma 8. *For $i \in \{1, \ldots, X-1\}$, if HI_i holds, then $\forall p \in R, \forall j < i+1$, $p.guest$ is equal to $\mathit{LLabels}(p)[j]$ 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 $\mathit{LLabels}(p)[1] = p.id = p.guest$ in phase 1. Hence the lemma holds for $i = 1$. Now assume that $i > 1$. Using HI_i, we have that for every $1 \leq j < i$, $\mathit{LLabels}(p)[j] = p.guest$ at phase j.

We consider now the case when $j = i$. Note that a process can only change the value of its variable $guest$ with Action B6, B8 or B9, namely during phase switching. Let p be a process at phase i and consider, in the execution, the step when p switches from phase $(i-1)$ to phase i: it 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 Actions B5 and B8). From Observation 1, 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 (Action B6, B8 or B9): hence, at phase i, $p.guest = \mathit{LLabels}(q)[i-1] = \mathit{LLabels}(p)[i]$. ■

From Observation 1, 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 8, we deduce the following corollary.

Corollary 9. *For $i \in \{1, \ldots, X-1\}$, if HI_i holds, then $\forall p \in R$, if p exits phase $j \leq i$ by Action B9, then $\mathit{LLabels}(p)[j]$ equals $p.id$.*

Lemma 10. *For $i \in \{1, \ldots, X-1\}$, if HI_i holds, then no action B9 is executed before phase $i+1$.*

Proof: Assume by contradiction that HI_i holds and some action B9 is executed before phase $i+1$. Consider the first time it occurs, say some process p executes Action B9 in some phase $j \leq i$.

From Corollary 9, by Action B9, p receives a message $\langle \text{PHASE\_SHIFT}, x \rangle$ with $x = p.id = \mathit{LLabels}(p)[j]$.

Furthermore, we have that $p.outer = k$ at phase j. Hence $p.id$ was observed $(k+1)$ times since the beginning of the execution: $p.guest$ took k times value $p.id$ and the value x in the received message is also $p.id$. By Lemma 8, the sequence of values of $p.guest$ is equal to $\mathit{LLabels}(p)^{j-1}$. Adding $x = \mathit{LLabels}(p)[j]$ at the end of the sequence, we obtain $\mathit{LLabels}(p)^j$. Hence, $j = \min\{x : \mathit{LLabels}(p)^x \text{ contains } p.id \ (k+1) \text{ times}\}$ and $n < j$ (this implies that $j \geq 2$, hence $(j-1) \geq 1$).

As p executes Action B9 in phase j, it is active during its whole j^{th} phase and hence exits its phase $(j-1)$ using Action B6. 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 12). We start by showing the following intermediate result:

Lemma 11. *While a process is in state* COMPUTE *(resp.* SHIFT*), the next message it has to consider cannot be of the form $\langle \text{PHASE\_SHIFT}, \_ \rangle$ (resp. $\langle x \rangle$).*

Proof: Assume by contradiction that some process p is in state COMPUTE (resp. SHIFT), but receives an unexpected message $\langle \text{PHASE\_SHIFT}, \_ \rangle$ (resp. $\langle x \rangle$) meanwhile. We examine the first case, the other case being similar. The unexpected message was transmitted through passive processes to p, but first initiated by some active process q (Action B5).

Since Action B5 was enabled at process q, q received k messages $\langle q.guest \rangle$ during one and the same phase. By the multiplicity, at least one of those messages, say m, was initiated by q using Action B1 or B6. So, m traversed the entire ring (Actions B2-B5, B7). Observation 1 ensures that this traversal occurs during one and the same phase. As a consequence, $q.guest \geq r.guest$ for every process r that were active when receiving m. In particular, $q.guest \geq p.guest$.

As q executed Action $B5$, k messages $\langle q.guest \rangle$ were sent by q (one action, either B1 or B6, and $(k-1)$ actions B3) 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, $p.guest \geq q.guest$ 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 Action B5 before receiving the unexpected message, a contradiction. ■

Lemma 12. *For every $i \in \{1, \ldots, X-1\}$, if HI_i holds, then $\forall p \in R$, 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 10 since HI_i holds, p cannot take state WIN before phase $(i+1)$. Hence, it cannot take state HALT by Action B9 or B11 as well. As no action B9 is executed during phase i, no message $\langle \text{FINISH}, \_ \rangle$ circulates in the ring during this phase (Observation 1): Action B10 cannot be enabled, hence p cannot take state HALT by Action B10. If p is in state COMPUTE (resp. SHIFT), it cannot receive any message $\langle \text{PHASE\_SHIFT}, \_ \rangle$ (resp. $\langle x \rangle$) by Lemma 11. Moreover, it cannot have received any message $\langle \text{FINISH}, \_ \rangle$ since no such message was sent during this phase (see Lemma 10 which applies as HI_i holds). As a conclusion, there is no way for p to deadlock during phase i. ■

Lemma 13. *For every $i \in \{1, \ldots, X-1\}$, if HI_i holds, then $\forall p \in R$, $\forall j < i+1$, $p \in Act_j$ if and only if p exits its phase j by Action B6 or B9.*

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 message $\langle LLabels(p)[i] \rangle$ (resp. any message) 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 Action B1 and sending its label $p.id = LLabels(p)[1]$. Otherwise ($i > 1$), by Lemma 10, no process can execute Action B9 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 Action B6 and sending its label $p.guest = LLabels(p)[i]$ (Lemma 8). By HI_i, if p is not in Act_{i-1}, p does not exits phase $(i-1)$ by executing Action B6 and so it cannot initiates a message with its label at the beginning of phase i.

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

Proof of Claim 2: Consider a message $m = \langle LLabels(L)[i] \rangle$ that circulates the ring (one exists 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 Observation 1) all along its ring traversal. From HI_i and Lemma 12, no process is deadlocked before its phase $(i+1)$. Hence, when m reaches a process in state PASSIVE, it is forwarded (Action B7) and when m reaches a process q in state COMPUTE (with $q.guest = LLabels(q)[i] \geq LLabels(L)[i]$, by Lemma 8 and definition of L), it is also forwarded unless Action B5 is enabled at q. This occurs at q if $LLabels(q)[i] = LLabels(L)[i]$, since $q.inner$ is initialized to 1 at the beginning of the phase (Action B1 or B6) and incremented if q receives $LLabels(q)[i]$. Hence, q has received k messages $\langle LLabels(L)[i] \rangle$ 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 $LLabels(q)[i] = LLabels(q')[i] = LLabels(L)[i]$, circulates during phase i, k messages $\langle LLabels(L)[i] \rangle$; any process between q and q' has forwarded them (and so received them).

Conclusion of the Proof: By HI_i, the lemma holds for all $j < i$. Let now consider the case $j = i$.

If $p \in Act_i$, then $LLabels(p)^i = LLabels(L)^i$ and in particular, $LLabels(p)[i] = LLabels(L)[i]$. As $Act_i \subseteq Act_{i-1}$, p is active at the end of phase $(i-1)$ and as no action B9 can take place before phase $(i+1)$ (Lemma 10), p is in state COMPUTE during the computation of phase i. Since $p.guest = LLabels(L)[i] \leq LLabels(q)[i]$ for any $q \in Act_{i-1}$ (Lemma 8, definition of L), and as any message $\langle x \rangle$ that circulates during the phase is initiated by some process $q \in Act_{i-1}$ with $x = LLabels(q)[i]$ (HI_i and Claim 1), p never executes Action B4 during phase i. Furthermore, p receives k times $p.guest$ during the phase (Claim 2), hence it

executes Action B5 followed by Action B6 or B9 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 $(i-1)$ with Action B8; it remains in state PASSIVE all along phase i and can only exit phase i with Action B8. Otherwise, $p \in Act_{i-1}$, *i.e.,* $LLabels(p)^{i-1} = LLabels(L)^{i-1}$ but $LLabels(p)[i] > LLabels(L)[i]$. p executes B4 at least when receiving the first occurrence of $\langle LLabels(L)[i] \rangle$ (Claim 2) and takes state PASSIVE. Once p is passive, it remains so and can only exit phase i using Action B8.

Finally, at least L executes Action B5: hence phase switching actually occurs (started by L or some other process) and causes every process to exit phase i. ■

This ends the proof of Lemma 7.

Theorem 3. B_k *solves the leader election for* $\mathcal{A} \cap \mathcal{K}_k$.

Proof: By Lemma 7 and by definition of X, no process is deadlocked before phase X and L is the only process that exits phase X executing Action B6 or B9. Now, by Lemma 7 and Corollary 9, $\forall i \in \{1, \ldots, X\}$, $L.guest = LLabels(L)[i]$ during phase i. Hence, when p begins its X^{th} phase, it is the $(k+1)^{\text{th}}$ time that L 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 Action $B9$. So, L sends a message $\langle \text{FINISH}, L.id \rangle$. L also sets $L.isLeader$ and $L.leader$ to TRUE and $L.id$, respectively. Every other process p receives the message in phase X (Observation 1) while being in state PASSIVE, since p exits its $(X-1)^{\text{th}}$ phase executing Action B8 (Lemma 7). So, p saves $L.id$ in its variable *leader*, then transmits the message to its right neighbor, and finally halts (Action B10). Finally, L receives $\langle \text{FINISH}, L.id \rangle$ and halts (Action B11). ■

Theorem 4. B_k *has time complexity* $O(k^2n^2)$*, message complexity* $O(k^2n^2)$*, 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 $O((k+1)n)$ time units. There is exactly X phases and $X \leq (k+1)n$. Thus, the time complexity of B_k is $O(k^2n^2)$.

During the first phase, every process starts by sending its *id*. Since a phase involves $O((k+1)n)$ actions per process, each process forwards labels $O((k+1)n)$ times. Finally, to end the first phase, every process sends and receives $\langle \text{PHASE\_SHIFT}, \_ \rangle$. Hence, $O(kn^2)$ 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 $d = mlty[\min\{p.guest : p \in Act_{i-1}\}]$. When phase i starts, every active process (at most k) sends its new $guest$. When the first message ends its first traversal ($O(kn)$ 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 turn of ring by a message. So the remaining messages (at most d) do at most $\frac{k}{d}$ traversal (n hops): $O(kn)$ messages. Overall, the phase requires $O(kn)$ messages exchanged. As there is at most $O(kn)$ phases, there are at most $O(k^2n^2)$ messages exchanged.

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

REFERENCES

[1] D. Angluin, "Local and global properties in networks of processors," in *STOC*, 1980, pp. 82–93.

[2] S. Kutten, G. Pandurangan, D. Peleg, P. Robinson, and A. Trehan, "Sublinear bounds for randomized leader election," in *ICDCN*, 2013, pp. 348–362.

[3] N. Lynch, *Distributed Algorithms*. Morgan Kaufmann, 1996.

[4] S. Dobrev and A. Pelc, "Leader election in rings with nonunique labels," *Fundam. Inform.*, vol. 59, no. 4, pp. 333–347, 2004.

[5] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, A. Kermarrec, E. Ruppert, and H. Tran-The, "Byzantine agreement with homonyms," *Dist. Comp.*, vol. 26, no. 5-6, pp. 321–340, 2013.

[6] D. Chaum and E. van Heyst, "Group signatures," in *EUROCRYPT*, 1991, pp. 257–265.

[7] S. Arévalo, A. F. Anta, D. Imbs, E. Jiménez, and M. Raynal, "Failure detectors in homonymous distributed systems (with an application to consensus)," *JPDC*, vol. 83, pp. 83–95, 2015.

[8] P. Flocchini, E. Kranakis, D. Krizanc, F. L. Luccio, and N. Santoro, "Sorting and election in anonymous asynchronous rings," *JPDC*, vol. 64, no. 2, pp. 254–265, 2004.

[9] C. Delporte-Gallet, H. Fauconnier, and H. Tran-The, "Leader election in rings with homonyms," in *NETYS*, 2014, pp. 9–24.

[10] K. Altisen, A. K. Datta, S. Devismes, A. Durand, and L. L. Larmore, "Leader election in rings with bounded multiplicity," in *SSS*, 2016, to appear.

[11] G. Tel, *Introduction to distributed algorithms (2nd Ed.)*. Cambridge University Press, 2000.

[12] R. C. Lyndon, "On burnsides problem," *Trans. of the AMS*, vol. 77, no. 2, pp. 202–215, 1954.

APPENDIX

A. Phase Numbering

The phases of each process p are defined according to the assignments on its variable $p.guest$. When p initializes $p.guest$ to $p.id$ in Action B1, p starts its first phase. Then, an active (resp. a passive) process p switches from some phase i to phase $i+1$ by setting $p.guest$ to the value x upon the reception of some message $\langle \text{PHASE\_SHIFT}, x \rangle$ in Action B6 or B9 (resp. Action B8).

B. Proof of Observation 1

We prove that when a given message is sent by a process and received by its right neighbor, those processes are at the same phase. The proof is split into three cases (Lemmas 14-16), according to the message types. Observation 1 is then a direct consequence of those lemmas.

Lemma 14. *Let p be a process, q its right neighbor, $m = \langle \text{PHASE\_SHIFT}, \_ \rangle$ a message, and $i, j \geq 1$.*

If m is sent by p in phase i and received by q in phase j, then $i = j$.

Proof: We prove the lemma by induction on $i \geq 1$.

Assume $i = 1$, p sends m during its first phase and q receives m at phase $j \geq 1$. Between two consecutive sendings of $\langle \text{PHASE\_SHIFT}, \_ \rangle$ (Action B5 or B8), p necessarily updates $p.guest$ (Action B6, B8, or B9), incrementing then its phase number. So m is necessarily the first $\langle \text{PHASE\_SHIFT}, \_ \rangle$ message p sent during the execution. Since q cannot receive messages from another process than p, q does not receive any other $\langle \text{PHASE\_SHIFT}, \_ \rangle$ message before m, so q is in its first phase upon the reception of m: $j = 1$.

Assume now that the lemma holds for $i \geq 1$. Assume also that p sends m during its phase $i+1$ and q receives it in phase $j \geq 1$. Consider the moment where p sent its previous $\langle \text{PHASE\_SHIFT}, \_ \rangle$ message, say m'. Again, between two consecutive sendings of $\langle \text{PHASE\_SHIFT}, \_ \rangle$ (Action B5 or B8), p necessarily updates $p.guest$ (Action B6, B8, or B9), incrementing then its phase number. So p was at most in phase i when p sent m'. Furthermore, p cannot update $p.guest$ two times (Action B6, B8, or B9) without sending some $\langle \text{PHASE\_SHIFT}, \_ \rangle$ message (Action B5 or B8) in between. (Notice also that a process cannot execute Action B9 twice.) Hence, p was in its phase i when it sent m'.

Now by induction hypothesis, q received m' in its i^{th} phase (Action B6 or B8): this started its phase $i+1$. Since it cannot receive messages from another process than p, q is in its $j = (i+1)^{\text{th}}$ phase when it receives m. ■

Lemma 15. *Let p be a process, q its right neighbor, $m = \langle x \rangle$ a message (where x a label), and $i, j \geq 1$.*

If m is sent by p in phase i and received by q in phase j, then $i = j$.

Proof: Assume by contradiction that the lemma is wrong and consider, without loss of generality, the first time this contradiction occurs during the execution: some message $m = \langle x \rangle$ was sent by a process p in phase i, received by its right neighbor q in phase j, but $i \neq j$.

If $j < i$ then $i > 1$ and p receives a message $\langle \text{PHASE\_SHIFT}, \_ \rangle$ (Action B6, B8, or B9), increments its phase number, and does not forward it (Action B5 or B8) to q before sending m. Precisely, p receives a message $\langle \text{PHASE\_SHIFT}, \_ \rangle$ either by executing Action B6 (but it necessarily executes Action B5 beforehand), or by executing Action B8: in both cases, p sends a message $\langle \text{PHASE\_SHIFT}, \_ \rangle$ to q before sending m, a contradiction.

Otherwise, $i < j$. Then, $j > 1$ and q receives a message $m' = \langle \text{PHASE\_SHIFT}, \_ \rangle$ from p that makes q switch from its $(j-1)^{\text{th}}$ to its j^{th} phase (Action B6, B8, or B9). By Lemma 14, p is in phase $j-1$ when it sends m'. Hence $i \geq j-1$, which in turns implies that $i = j-1$: p sends m' and then m without incrementing its phase number (Action B6, B8, or B9) in between. As a consequence, either p sends m' by Action B5 and p necessarily sends m by Action B6, or p sends m' by Action B8. In both cases, p updates $p.guest$ and so increments its phase number before sending m, a contradiction. ■

Lemma 16. *Let p be a process, q its right neighbor, $\langle \text{FINISH}, \_ \rangle$ a message, and $i, j \geq 1$.*

If m is sent by p in phase i and received by q in phase j, then $i = j$.

Proof: There are two cases.

1) If p sends m by Action B9, then p switches from phase $i-1$ to i, and p necessarily executes Action B5 beforehand. So q receives $m' = \langle \text{PHASE\_SHIFT}, \_ \rangle$ from p just before m. By Lemma 14, p sends m' in phase $i-1$ so q receives m' in phase $i-1$ and switches to phase i. Now, q cannot assign $q.guest$ (and so increment its phase number) between the reception of m' and m. So q receives m in phase $j = i$.

2) If p sends m executing Action B10, then beforehand, p necessarily executes Action B4, B7, or B8. If p executes Action B8 beforehand, using similar arguments as in case 1), q receives m in phase $j = i$. Otherwise, p

sends a message $m' = \langle x \rangle$, where x is a label, in phase i (p cannot update $p.guest$ and increment its phase number in between) to q. By Lemma 15, q receives m' in phase i and cannot update $q.guest$ (and increment its phase number) between the reception of m' and m. Hence, q receives m in phase $j = i$. ■

C. Proof of Corollary 9

Corollary 9. *For $i \in \{1, \ldots, X-1\}$, if HI_i is true, then $\forall p \in R$, if p exits phase $j \leq i$ by Action B9, then LLabels$(p)[j]$ equals $p.id$.*

Proof: Let $i \in \{1, \ldots, X-1\}$. Assume that HI_i holds. Let p be a process that executes Action B9 in some phase $j \leq i$. By this action, p receives a message $m = \langle \text{PHASE\_SHIFT}, x \rangle$. Note that, since $k = p.outer \leq j$ in phase j and since $k \geq 2$, we have $j > 1$.

p receives m from its left neighbor, q, x being the value of $q.guest$ when q sent m (Action B8 or B5). From Lemma 14, and since p receives m at phase j, q also sends m in phase j. Hence, $x = q.guest$ in phase $j-1$. Hence, $x = LLabels(q)[j-1] = LLabels(p)[j]$. ■