From Bezout's Identity to Space-Optimal Election in Anonymous Memory Systems

Emmanuel Godard
LIS, Aix-Marseille University & CNRS & Univ. Toulon
Marseille, France

Damien Imbs
LIS, Aix-Marseille University & CNRS & Univ. Toulon
Marseille, France

Michel Raynal
Univ Rennes, INRIA, Cnrs, IRISA, France
& Hong Kong Polytechnic University

Gadi Taubenfeld
The Interdisciplinary Center
Herzliya, Israel

ABSTRACT

An anonymous shared memory *REG* can be seen as an array of atomic registers such that there is no a priori agreement among the processes on the names of the registers. As an example a very same physical register can be known as $REG[x]$ by a process p and as $REG[y]$ (where $y \neq x$) by another process q. Moreover, the register known as $REG[a]$ by a process p and the register known as $REG[b]$ by a process q can be the same physical register. It is assumed that each process has a unique identifier that can only be compared for equality. This article is on solving the *d-election* problem, in which it is required to elect at least one and at most d leaders, in such an anonymous shared memory system. We notice that the 1-election problem is the familiar leader election problem. Let n be the number of processes and m the size of the anonymous memory (number of atomic registers). The article shows that the condition $\gcd(m, n) \leq d$ is necessary and sufficient for solving the d-election problem, where communication is through read/write or read+modify+write registers. The algorithm used to prove the sufficient condition relies on *Bezout's Identity* – a Diophantine equation relating numbers according to their Greatest Common Divisor. Furthermore, in the process of proving the sufficient condition, it is shown that 1-leader election can be solved using only a *single* read/write register (which refutes a 1989 conjecture stating that three non-anonymous registers are necessary), and that the *exact* d-election problem, where exactly d leaders must be elected, can be solved if and only if $\gcd(m, n)$ divides d.

CCS CONCEPTS

• **Theory of computation → Distributed computing models**; **Shared memory algorithms**; **Distributed algorithms**.

KEYWORDS

Anonymous register, Asynchrony, Atomic register, Bezout Identity, Bounded register, Concurrent algorithm, Equality-based comparison, Leader election, Process identity, RW register, Symmetric algorithm, Symmetry breaking.

ACM Reference Format:
Emmanuel Godard, Damien Imbs, Michel Raynal, and Gadi Taubenfeld. 2020. From Bezout's Identity to Space-Optimal Election in Anonymous Memory Systems. In *Symposium on Principles of Distributed Computing (PODC '20), August 3–7, 2020, Virtual Event, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3382734.3405727

1 INTRODUCTION

1.1 Problems and results

Leader election. *Leader election* is a classical fundamental problem encountered in distributed computing. Each process p_i has a local variable $leader_i$ initialized to a default value (e.g., $\perp$). The leader election problem is to an algorithm at the end of which the local variables $leader_i$ forever contain the same identity, which is the identity of one of the participating processes. A classical way to elect a leader is select the process with the smallest (or greatest) identity.

The leader election problem has been intensively investigated in asynchronous failure-free systems where the processes communicate by message-passing. In this context the aim was to obtain algorithms whose message complexity is optimal [1].

In the context where the processes communicate through atomic shared registers, two types of election algorithms have been introduced in [19]. One in which all the processes are required to participate (i.e., to compete to be a leader), the other in which any subset of processes can participate (but not necessarily all the processes). In this context, the following results are shown in [19] (n is the number of processes):

- $\lceil \log n \rceil + 1$ atomic read/write (non-anonymous) registers are necessary and sufficient to elect a process as a leader if the participation of all the n processes is not required, and
- 3 atomic read/write (non-anonymous) registers are sufficient to elect a process as a leader when the participation of all the processes is required. Moreover, it is conjectured that three registers are also necessary.

[1] A survey on election algorithms in failure-free message-passing systems appears in Chapter 4 of [15].

PODC '20, August 3–7, 2020, Virtual Event, Italy

ACM ISBN 978-1-4503-7582-5/20/08…$15.00
https://doi.org/10.1145/3382734.3405727

If processes may crash, the system is fully asynchronous, and the elected leader must be a process that does not crash, leader election cannot be solved. Not only the system can no longer be fully asynchronous but the leader election problem must be weakened to the *eventual leader* election problem (called Ω in the failure detector parlance [7]). In this case the algorithm is required to automatically –i.e., without external intervention– elect a new leader each time the previously elected leader crashes[2].

d-election. The *d-election* problem, where $1 \leq d < n$, is a natural generalization of leader election. It consists in selecting at least one and at most d leaders. That is, each process writes some participating process identity into its local variable $leader_i$, and the size of the set of all identities written must be at most d. Furthermore, if some process writes the identity of process p into its local variable, then p must also do so. That is, each process knows whether it is elected or not. Leader election is the case where $d = 1$, in which each process elects a single leader and there is no disagreement among the processes on the elected leader. We will also consider another variation of the problem, which we call the *exact d-election* problem, in which exactly d leaders must be elected.

Main results. Considering asynchronous systems in which the processes communicate through anonymous (see below) atomic read/write (RW) or read+modify+write (RMW) registers, and all the processes are required to participate, the following results are presented in the paper. Let n be the number of processes and m the size of the anonymous memory (number of atomic registers). We show that the condition $\gcd(m, n) \leq d$ is necessary and sufficient for solving the d-election problem, where communication is using either RW or RMW registers. The algorithm used to prove the sufficient condition relies on *Bezout's Identity*, a Diophantine equation relating pairs of integers according to their Greatest Common Divisor. Furthermore, in the process of proving the sufficient condition, it is shown (i) that leader election can be solved in systems where communication is through a *single* (bounded) read/write register, thereby refuting the 1989 conjecture stating that three (non-anonymous) registers are necessary [19], and (ii) that the *exact d*-election problem can be solved using either read/write or RMW registers if and only if $\gcd(m, n)$ divides d.

1.2 The computational model

Process model. The system is composed of a finite set of $n \geq 2$ asynchronous sequential processes denoted $p_1, .., p_n$. The subscript i in p_i is only a notational convenience, which is not known by the processes. Each process p_i knows its unique identity id_i and the total number of processes n. The set of possible process identifiers is larger than n and is not a priori know by the processes.

No process fails and each process participates in the algorithm. *Asynchronous* means that each process proceeds to its own speed, which can vary with time and always remains unknown to the other processes.

Anonymous memory. While *process anonymity* has been studied for a long time from an algorithmic and computability point of view, both in message-passing systems (e.g., [2, 4, 22]) and shared memory systems (e.g., [3, 5, 9]), the notion of *memory anonymity* has been introduced only very recently [21].

Let us consider a shared memory REG made up of m atomic registers. Such a memory can be seen as an array with m entries, namely $REG[1..m]$. In a non-anonymous memory system, for any index x, $1 \leq x \leq m$, if two or more processes invoke the address $REG[x]$ they access the very same register. As stated in [21], in the classical system model, there is an a priori agreement on the names of the shared registers. This a priori agreement facilitates the implementation of the coordination rules the processes have to follow to progress without violating the safety (consistency) properties associated with the application they solve [15, 20].

This a priori agreement no longer exists in a memory-anonymous system. In such a system the very same address identifier $REG[x]$ invoked by a process p_i and invoked by a different process p_j does not necessarily refer to the same atomic register. More precisely, a memory-anonymous system with m registers is such that:

- for each process p_i, the adversary defines a permutation $f_i()$ over the set $\{1, 2, \cdots, m\}$, such that when p_i addresses $REG[x]$, it actually accesses $REG[f_i(x)]$, and
- no process knows the permutations.

It is assumed that all the registers are initialized to the same value. Otherwise, thanks to their different initial values, it would be possible to distinguish different registers, and consequently the registers would no longer be fully anonymous. Finally, we point out that, unlike in the case of non-anonymous registers, the fact that a problem is solvable using m anonymous registers, does not imply that it is also solvable using $m + 1$ anonymous registers [21].

RW and RMW communication models. This article considers two types of communication models. In the first one the processes communicate through atomic read/write (RW) registers only. Hence, any register can be atomically read or written by any process. In the second communication model, in addition to read and write operations, a process can access any register with a stronger atomic read/modify/write (RMW) operation. An example of such an atomic RMW operation is the well-known compare&swap operation. Denoted `compare&swap`(X, old, new), this operation has three input parameters, a shared register X and two values old and new. It returns a Boolean value. It has the following effect: if $X = old$ the value new is assigned to X and the value `true` is returned (the `compare&swap()` operation is then successful). If $X \neq old$, Z is not modified, and the value `false` is returned. The paper uses the RW model for algorithms, and the stronger RMW model for impossibility proofs. Let us notice that the registers of a memory-anonymous system are necessarily multi-writer multi-reader registers.

Atomic means that the operations on the registers appear as if they have been executed sequentially [10], each operation appearing between its start event and its end event, and for any $x \in \{1, ...m\}$, each read operation of a register $R[x]$ returns the value v, where v is the last value written in $R[x]$ by a write or the considered RMW operation (in the case of the RMW communication model).

Symmetry constraint on process identities. This paper considers that process identities can only be compared for equality (there is

[2]Surveys on eventual leader election algorithms in asynchronous crash-prone systems appear in Chapter 17 of [14] when communication is through a shared memory, and Chapter 18 of [16] when communication is through message-passing.

no notion of "greater than" or "smaller than" on process identities). More precisely, process identities define a programming type the values of which can be read, written and compared for equality only. This requirement has been introduced in [19] and investigated in [21] where it is called *symmetry*. An algorithm satisfying it is said to be *symmetric*. This constraint is motivated by the following reasons.

- Design algorithms as general as possible. As they do not need to rest on an order relation on process identities, symmetric algorithms require fewer assumptions and are consequently more general than non-symmetric algorithms.
- Electing a process with an extremal identity value entails the election of the same process in all the executions of a non-symmetric algorithm. In this sense, the symmetry constraint on process identities can be seen as a "last step" before process anonymity.
- More generally, symmetric means "egalitarian". The fact that a process is elected does not rely on its singularity dimension, namely its identity or a specific code it has to execute. The result of an election only depends on the run, i.e., on the specific asynchrony pattern exhibited by the current run (which is independent of the identities the processes are provided with).

1.3 Motivation, content and related work

Motivation. This work has two primary motivations. The first is related to the basics of computing, namely, computability and complexity lower/upper bounds. Increasing our knowledge of what can (or cannot) be done in the context of anonymous memories, and providing associated necessary and sufficient conditions, helps us determine the weakest system assumptions under which fundamental problems, such as mutual exclusion or leader election can be solved. The second one is application-oriented. It appears that the concept of an anonymous memory allows epigenetic cell modification to be modeled from a computing point of view [13]. Hence, anonymous shared memories could be useful in biologically inspired distributed systems [11, 12]. If this is the case, mastering leader election in such an adversarial context could reveal to be important from an application point of view.

Content & roadmap. This article addresses symmetric d-election in a fault-free anonymous shared memory system made up of $m \geq 1$ atomic registers and n processes, all participating in the algorithm. It presents the following results.

- A necessary and sufficient condition for d-election, relating the number of processes n and the size of the anonymous memory m, be the atomic registers either RW or RMW, namely $\gcd(m, n) \leq d$.
- An algorithm, based on Bezout's Identity, that uses RW registers and solves the exact d-election problem when $\gcd(m, n)$ divides d, and an impossibility proof that exact d-election cannot be solved if $\gcd(m, n)$ does not divide d, even if the registers are RMW registers.

The necessary conditions are proved in Section 2. The algorithms for leader election and exact d-election when $m \geq 1$ are built incrementally. In Section 3, a first algorithm is presented which elects a leader on top of a single bounded atomic RW register. In the specific case of a memory made up of a single register, the single register is not anonymous. Hence, as already said, the very existence of this algorithm refutes the conjecture stated in [19] on the number of atomic read/write registers needed to solve 1-election when the participation of all the processes is required. Not only the algorithm for 1-election uses a single read/write register, but this register is *bounded*, namely, assuming $\log_2 n$ bits are needed to encode a process identity, the size of the register is $1 + n \log_2 n$ bits.

Then, in Sections 4 and 5, using Bezout's Identity, the previous algorithm is generalized to obtain an exact d-election algorithm for a shared memory made up of $m \geq 1$ atomic RW anonymous registers. This algorithm also uses a bounded number of bounded-size registers only.

Finally, Section 6 combines the necessity result of Section 2 with the algorithm of Section 5 to obtain tight bounds for leader election, d-election and exact d-election.

Related work on anonymous memory. The work described in [21], that introduced the notion of an anonymous memory, addressed mutual exclusion (in short mutex), consensus, election and renaming, problems for which it presented algorithms and impossibility results. The consensus, election and renaming algorithms presented in [21] satisfy the obstruction-freedom progress condition, namely, if a process executes alone during a long enough period, it eventually decides.

Two symmetric deadlock-free mutual exclusion algorithms for anonymous memory systems are presented in [1]. The first considers the RW communication model. It assumes that $m \in M(n) \setminus \{1\}$ where $M(n) = \{m : \forall\, \ell : 1 < \ell \leq n :\ \gcd(\ell, m) = 1\}$. The second considers the RMW communication model. It assumes that $m \in M(n)$. It is shown in [1, 21] that each of these conditions is necessary for the corresponding communication model.

When the participation of all the processes is not required, any mutex algorithm can be used to elect a leader. In the case of a non-anonymous memory, it is shown in [19] that n atomic read/write registers are necessary and sufficient for symmetric deadlock-free mutex[3]. The impossibility proof for mutex presented in [1] can be trivially adapted to show that, when mutex is impossible, 1-election without required participation is also impossible. When considering anonymous registers, it thus appears that the previous necessary and sufficient conditions for symmetric deadlock-free mutex ($m \in M(n) \setminus \{1\}$ in the RW model and $m \in M(n)$ in the RMW model) applies to 1-election when not all processes are required to participate.

Leader election on top of anonymous atomic RW registers has been addressed in [8], which describes three algorithms suited to particular cases, namely, $m = \alpha \times n + 1$, $m = \alpha \times n + (n - 1)$, and $m \in M(n) \setminus \{1\}$. Differently from the general algorithm presented in this article, these three algorithms are different, each leveraging the specific value of m for which it solves the leader election problem.

[3] This result extends a previous result from [6] stating that n atomic read/write registers are necessary and sufficient for asymmetric deadlock-free mutex in a non-anonymous memory.

Finally, fully anonymous shared memory systems, where *both* processes and memory are anonymous, were recently investigated in [17, 18].

2 A NECESSARY CONDITION FOR d-ELECTION IN AN ANONYMOUS MEMORY

This section presents an impossibility result relating the number of processes (n) and the size of the anonymous memory (m). To make the result as strong possible, it considers the following assumptions.

- Communication is via atomic RMW registers. Thus, the result also holds for RW registers.
- Upon termination, each process needs to know only if it is a leader or not. More precisely, each process has a write-once private output register which is set to 0 if the process is not elected and to 1 if it is elected.

Theorem 1 (**Impossibility Results**).

(1) *There is* no symmetric d-election algorithm *for* $n \geq 2$ *processes using* m *anonymous* RMW *registers when* $\gcd(m, n) > d$.
(2) *There is* no symmetric exact d-election algorithm *for* $n \geq 2$ *processes using* m *anonymous* RMW *registers if* $\gcd(m, n)$ *does not divide* d.

Proof. Assume $\gcd(m, n) = \delta$. Let us divide the n processes into δ disjoint groups, denoted $P_0, ..., P_{\delta-1}$, such that there are exactly n/δ processes in each group. This division is achieved by assigning process p_i (where $i \in \{0, ..., n-1\}$) to group $P_{i \bmod \delta}$. For example, for six processes and $\delta = 3$, $P_0 = \{p_0, p_3\}$, $P_1 = \{p_1, p_4\}$, and $P_2 = \{p_2, p_5\}$. Such a division is possible since $\gcd(m, n) = \delta$.

Let us arrange the m registers on a ring with m nodes where each register is placed on a different node. To each one of the δ groups of processes P_i (where $i \in \{0, ..., \delta - 1\}$), let us assign an initial register (namely, the first register that each process in that group accesses) such that for every two groups P_i and its ring successor $P_{(i+1) \bmod \delta}$ the distance between their assigned initial registers is exactly m/δ when walking on the ring in a clockwise direction. This is possible since $\gcd(m, n) = \delta$.

The lack of global names allows us to assign for each process an initial register and an ordering which determines how the process scans the registers. An execution in which the n processes are running in *lock-steps*, is an execution where we let each process take one step (in the order $p_0, ..., p_{n-1}$), and then let each process take another step, and so on. For a given d-election algorithm A, let us call this execution, in which the processes run in lock-steps, ρ_A. For simplicity, we will omit the subscript A and simply write ρ.

For process p_i and integer k, let $\text{order}(p_i, k)$ denotes the k^{th} new register that p_i accesses during the execution ρ, and assume that we arrange that $\text{order}(p_i, k)$ is the register whose distance from p_i's initial register is exactly $(k-1)$, when walking on the ring in a clockwise direction.

We notice that $\text{order}(p_i, 1)$ is p_i's initial register, $\text{order}(p_i, 2)$ is the next new register that p_i accesses and so on. That is, p_i does not access $\text{order}(p_i, k+1)$ before accessing $\text{order}(p_i, k)$ at least once, but for every $j \leq k$, p_i may access $\text{order}(p_i, j)$ several times before accessing $\text{order}(p_i, k+1)$ for the first time. (When a process accesses a register for the first time, say register $REG[x]$, we may map x to any (physical) register that it hasn't accessed yet. However, when it accesses $REG[x]$ again, it must access the same register it has accessed before when referring to x.)

Next, let us consider another division of the n processes into groups. We divide the n processes into n/δ disjoint groups, denoted $Q_0, ..., Q_{n/\delta-1}$, such that there are exactly δ processes in each group. This division is achieved by assigning process p_i (where $i \in \{0, ..., n-1\}$) to group $Q_{\lfloor i/\delta \rfloor}$. For example, for six processes and $\delta = 3$, $Q_0 = \{p_0, p_1, p_2\}$, and $Q_1 = \{p_3, p_4, p_5\}$. Again, such a division is possible since $\gcd(m, n) = \delta$.

We notice that Q_0 includes the first process to take a step in the execution ρ, in each one of the δ groups, $P_0, ..., P_{\delta-1}$. Similarly, Q_1 includes the second process to take a step in the execution ρ, in each one of the δ groups, $P_0, ..., P_{\delta-1}$, and so on.

Since only comparisons for equality are allowed, and all registers are initialized to the same value – which (in order to preserve anonymity) is not a process identity – in the execution ρ, for each $i \in \{0, ..., n/\delta - 1\}$, all the processes in the group Q_i that take the same number of steps must be at the same state. Thus, in the run ρ, it is not possible to break symmetry within a group Q_i ($i \in \{0, ..., n/\delta - 1\}$), which implies that either all the δ processes in the group Q_i will be elected, or no process in Q_i will be elected.

Thus, the number of elected leaders in ρ equals δ times the number of Q_i groups ($i \in \{0, ..., n/\delta - 1\}$) that all their members were elected, and (by definition of d-election) it must be a positive number. That is, the number of elected leaders in ρ equals $a\delta$ for some integer $a \in \{1, ..., n/\delta\}$. We consider two cases: (1) Since at most d leaders are elected, it follows from the fact that for some positive integer a, $a\delta \leq d$ that, for both d-election and exact d-election, $\gcd(m, n) = \delta \leq d$. This completes the proof of the first part of the theorem; (2) Since, in the case exact d-election, exactly d leaders are elected, it must be the case that, $d = a\delta$ for some integer $a \in \{1, ..., n/\delta\}$. This completes the proof of the second part of the theorem. □ *Theorem 1*

3 LEADER ELECTION ON TOP OF A SINGLE READ/WRITE ATOMIC REGISTER

This section presents Algorithm 1, which elects a leader in an n-process asynchronous system, where communication is through a single bounded atomic RW register. Hence, it can be used in a non-anonymous memory as well. As already said, and as far as we know, it is the first RW election algorithm in which the processes communicate through a single (bounded) RW register.

3.1 Presentation of the algorithm

Algorithm 1 is based on the principle of a dialog between candidates for leader and confirmed losers. A candidate p_i tries to impose its identity by writing it in (a field of) the memory (it may later be overwritten by another candidate). A process p_j that observes such write loses; it informs the remaining candidates of its loss by writing its identity in another field of the memory. Subsequently, a write by a candidate can only be triggered by a write by a confirmed loser, and conversely. When a candidate has received an acknowledgment from every other process, it is elected. As we will see, in order to ensure that all the losers know the identity of the leader, processes

```
init of REG.id ← ⊥; REG.discarded ← ∅; REG: REG.phase ← electing.

operation election(id_i) is   % code for process p_i, i ∈ {1, ··· , n}
      phase_i ← electing; discarded_i ← ∅; loser_i ← false; candidate_i ← true;
(1)   repeat
(2)       reg_i ← REG;
(3)       if ((reg_i.phase = committing) ∧ (phase_i = electing)) then
                % move to committing phase %
(4)             phase_i ← committing; discarded_i ← ∅
(5)       else if ((|reg_i.discarded| = n − 1) ∧ (reg.id ≠ ⊥))
                % only one winner remains; return or move to committing phase %
(6)             if (reg_i.phase = committing)
(7)                then return (reg.id)
(8)                else REG ← ⟨⊥, ∅, committing⟩
(9)             end if
(10)      else if ((reg_i.id = ⊥) ∧ ¬loser_i ∧ candidate_i) then
                % p_i is a potential winner; update REG.discarded %
(11)            if (reg_i.phase = phase_i)
(12)                  then discarded_i ← discarded_i ∪ reg_i.discarded
(13)            end if;
(14)            REG ← ⟨id_i, discarded_i, phase_i⟩
(15)      else if ((reg_i.id ∉ {⊥, id_i}) ∧ (id_i ∉ reg_i.discarded)) then
                % p_i has already lost; inform (potential) winner(s) %
(16)            loser_i ← true; discarded_i ← reg_i.discarded ∪ {id_i};
(17)            REG ← ⟨⊥, discarded_i, phase_i⟩
(18)      end if
(19)  end repeat.
```

Algorithm 1: Leader election for n processes using a single bounded atomic RW register *REG*

go through a similar second phase, during which there is a single candidate.

The shared register. This register, denoted *REG*, is composed of three fields: *REG.id* (which is problem-oriented), and *REG.phase* and *REG.discarded* (which are specific to the algorithm and are consequently implementation-oriented).

- *REG.id* contains a process identity or the default value $\perp$. It is initialized to $\perp$. At the end of the execution, it contains the identity of the elected process.
- *REG.phase* $\in$ {`electing`, `committing`} (phase name). It is initialized to `electing`.
- *REG.discarded* is a set containing process identities. It is initialized to $\emptyset$. At the end of the execution, it contains the identities of every process except the elected one.

Local variables at a process p_i. Each process p_i manages three local variables.

- reg_i: local copy of *REG*. It is consequently composed of three fields $reg_i.id$, $reg_i.phase$ and $reg_i.discarded$ initialized as *REG*.
- $phase_i$ (initialized to `electing`): current phase of p_i.
  - $phase_i$ = `electing` means that from p_i's point of view, no leader has yet been elected. Hence, processes are still competing to become the elected leader.
  - $phase_i$ = `committing` means that from p_i's point of view, a leader has been elected. Its identity is then the only one that can appear in *REG.id*.
- $loser_i$ is a write-once Boolean initialized to `false`. It is set to `true` by p_i when it discovers it is eliminated from the competition.
- $discarded_i$ is a set initialized to $\emptyset$. Its meaning depends on the current phase.
  - If $phase_i$ = `electing`, $id_j \in discarded_i$ means that p_i knows that p_j will not be elected.
  - If $phase_i$ = `committing`, $id_j \in discarded_i$ means that p_i knows that p_j knows the leader.
- $candidate_i$ is not used in the case of leader election. It will be used in Claim 2, in order to help extend the algorithm to exact d-election.

Description of the algorithm. Algorithm 1 is a two-phase-based algorithm. During the first phase, processes compete to impose their identity in *REG.id*. During the second phase, the leader ensures that all processes acknowledge that the election is over. The main difference between the two phases is that, in the first phase, various processes can act as potential leaders, while in the second phase, only a single process can have this behavior, thus ensuring that the memory remains stable at the end of the election.

Structure of the loop. After reading the register at line 2, p_i proceeds to check whether it is in one of the following cases.

- *The second phase already started.* Process p_i first checks whether it should change of phase (namely if the election has progressed to `committing` while itself is still in the `electing` phase, line 3). If so, it updates $phase_i$ to `committing` and resets $discarded_i$ to $\emptyset$ (line 4).
- *The phase is over.* If $|REG.discarded| = n-1$ (all processes except one have announced that they cannot win) and $REG.id \neq \perp$ (*REG.id* contains the identity of the leader), the current phase is over (line 5). If the phase is `committing`, process p_i returns the identity of the leader (line 7). Otherwise, it resets *REG.id* to $\perp$, *REG.discarded* to $\emptyset$ and sets *REG.phase* to `committing`.
- *p_i competes to be the leader.* If $REG.id = \perp$ and $loser_i$ = `false` (line 10), p_i tries to impose itself as the leader. The

predicate $reg_i.id = \perp$ means that either the current phase has just started (initial state of *REG* or write at line 8) or the previous write was by a process that has already lost (line 17). Process p_i then checks if this last write corresponds to its current phase (line 11), and if so, adds the content of $reg_i.discarded$ to $discarded_i$ (line 12). Otherwise, the previous writer hadn't observed the phase change yet, and p_i doesn't update $discarded_i$. It then writes its identity id_i, $discarded_i$ and the current phase $phase_i$ in *REG* (line 14).

- p_i *lost and must inform the leader.* If $reg_i.id \notin \{\perp, id_i\}$, another process p_j tries to be the leader, and p_i loses the competition. Additionally, if $id_i \notin reg_i.discarded$, the leader needs to receive an acknowledgment from p_i. It then sets $loser_i$ to `true`, includes its identity in $reg_i.discarded$ and informs the leader by writing accordingly in *REG*, setting *REG.id* to $\perp$ (lines 16-17).

Two phases. The first phase elects the leader, and the second phase ensures that all the processes have access to its identity. The second phase is necessary: during the first phase, a non-leader p_i may observe a write by a potential leader p_j that will ultimately lose, such that id_i is not included in $discarded_j$, and thus p_i is poised to write again. The real leader p_ℓ may have observed p_i's loss, and have ended the phase. In such a case, the write by p_i will erase all information about p_ℓ's write from the memory, preventing p_i from knowing that p_ℓ is the leader. During the second phase, all processes except p_ℓ act as losers, preventing such behavior, and thus allowing all processes to know the identity of the leader.

Termination. When, at the end of the `committing` phase, the leader has collected acknowledgments from all other processes (that have written at line 17), it writes its identity in *REG.id*, along with the acknowledgments in *REG.discarded* (line 14). The competition is then over: no process will write in the memory again, and the predicates of lines 5-6 are satisfied, allowing all the processes to return the identity of the leader (line 7).

3.2 Proof of the algorithm

Theorem 2. *Algorithm* 1 *is a* symmetric *algorithm that solves* leader election *in an n-process system where communication happens through* $m = 1$ *atomic* RW *register.*

Proof. Assuming all the processes invoke election(), let us first show that each process terminates.

Initially, *REG.phase* is set to `electing`, and the sets $discarded_i$ and *REG.discarded* are empty. Every process will then proceed to the *if* statements at lines 10 and 15. As initially $REG.id = \perp$, it follows from lines 10-14 that there is a finite time at which, for any process p_i that reads $\perp$ from *REG.id*, due to line 14 we have $REG.id = id_i$ (where the write of p_i may later be overwritten by another process identity). Moreover, after a process p_j reads $REG.id \notin \{\perp, id_j\}$, due to the predicate of line 15, it sets $loser_j$ to `true` (which is never modified thereafter) and resets *REG.id* to its initial value $\perp$ (line 17). Due to the "repeat" loop, it follows that this pattern is repeated until all the processes p_j except one, say p_ℓ, are such that $loser_j$ = `true`.

Then each loser process p_j executes lines 15-17: it writes $\perp$ in *REG.id* and includes its identity in *REG.discarded*. Once this occurred, p_ℓ is the only process that can execute lines 10-14, at which it adds the content of *REG.discarded* (which contains process identities) to its local set $discarded_\ell$ (line 12) and writes $discarded_\ell$ into *REG.discarded* (line 14). If p_j's write is overwritten by another loser process, the next write by p_ℓ will not include id_j in *REG.discarded*, and p_j will execute lines 15-17 again. Let us observe that the process identities in $discarded_\ell$ are not suppressed from it until the end of the first phase (which occurs when any process executes line 17). As each loser process repeatedly writes $\perp$ in *REG.id* and its identity in *REG.discarded*, it follows that $discarded_\ell$ eventually grows until it contains all the process identities except its own identity id_ℓ. When this occurs, the predicate of line 16 is true at p_ℓ, which writes the tag `committing` in *REG.phase*, resets *REG.discarded* to $\emptyset$, and leaves $REG.id = id_\ell$ (line 8).

The loser processes then enter the second phase in which, as their variables $loser_j$ are equal to `true`, they can only execute lines 16-17, while p_ℓ executes lines 10-14 each time it reads $REG.id = \perp$. The behavior of the processes is then similar to the one of the first phase, at the end of which *REG.discarded* eventually contains the identities of all the loser processes. It follows that eventually the predicates of lines 5 and 6 is satisfied, directing each process to executes line 7, which terminates its execution.

The proof of the safety property (a single process is elected) comes from the observation that, at the end of the first phase, *REG.id* contains the identity of a process p_ℓ and, as each other process p_j is a loser, due to the predicate of line 10, only p_ℓ can write its identity in *REG.id* at line 14. A loser process can only write $\perp$ in *REG.id* (line 17). Finally, as each time it contains $\perp$, *REG.id* is reset by p_ℓ to id_ℓ (lines 10-14), it follows that, when $REG.id \neq \perp$ at line 5 of the `committing` phase, it necessarily contains the identity of the elected process p_ℓ. □ *Theorem 2*

The following claims are not directly related to the properties of leader election. They will be used in the proof of Algorithm 2 (Section 5).

Claim 1. *After a process returns at line* 7*, no more writes to the memory can occur.*

Proof. Let p_ℓ be the elected leader, and p_i be another process. Note that, during the `committing` phase, p_ℓ is the only one that can execute lines 10-14 (every other process p_i has $loser_i$ set to `false`).

During a phase, no identity is removed from p_ℓ's $discarded_\ell$ set (line 12). Additionally, a write by p_i can only occur if it observes a write by p_ℓ such that $id_i \notin reg_i.discarded$ (lines 15-17). When p_i observes such a write, $id_i \notin discarded_\ell$, and no process can terminate. Thus no process can terminate before p_ℓ executes a write at line 14 with $|discarded_\ell| = n - 1$ and $phase_\ell$ = `committing`, and once it does, no process will write again in the memory. □ *Claim 1*

Claim 2. *If, for every process* p_i*,* $candidate_i$ *is initialized to* `false`*, no process will terminate. Additionally, if during the execution at least one process* p_j *sets* $candidate_j$ *to* `true`*, a single one of these processes will be elected, and all processes will terminate.*

Proof. If for every p_i, $candidate_i$ is initialized to `false`, the condition at line 10 cannot be satisfied. Moreover, due to the initial

values of the fields $REG.id = \perp$ and $REG.phase_i$ = `electing`, none of the predicates at lines 3, 5, and 15 is satisfied. It follows that no process will write, and consequently none of them will terminate.

If at least one process p_i sets $candidate_i$ to `true`, it will try to fill the role of leader (condition at line 10), and the reasoning of Theorem 2 applies. Furthermore, a process that didn't set $candidate_i$ to `true` cannot execute lines 10-14, and thus cannot be elected. □ *Claim 2*

4 FROM BEZOUT'S IDENTITY TO LEADER ELECTION AND d-ELECTION

Theorem 3. (Bezout, 1730-1783) *Let m and n be two positive integers such that $\gcd(m,n) = d$. There are two integers u and v such that $u \times m + v \times n = d$.*

The pair $\langle u, v\rangle$ is not unique. Euclid's gcd() algorithm can be used to compute such pairs.

Playing with Bezout's Identity. As m and n are positive, it follows that one of u and v is positive while the other is negative or null. As explained below, we are interested in the case where Bezout's Identity is rewritten as $u' \times m = v' \times n + d$ (where u' is positive and v' is positive or null).

- If v is negative or null, let u' be u and v' be $-v$, giving $u' \times m = v' \times n + d$.
- If u is null, then $n = \gcd(m,n)$ and divides m. Let $u' = 1$ and $v' = \frac{m}{n} - 1$.
- If u is negative, let $w = -u$. We have:
$$-w \times m = -v \times n + d$$
$$vwnm - w \times m = vwmn - v \times n + d$$
$$(vn-1) \times w \times m = (wm-1) \times v \times n + d$$
Let $u' = (vn-1) \times w$ and $v' = (wm-1) \times v$. Since m, n, v and w are all positive, we obtain $u' \times m = v' \times n + d$ where both u' and v' are positive.

It follows that, whatever the initial values of u and v obtained from Bezout's identity for the pair $\langle m, n\rangle$, it is always possible to replace them by two positive integers u' and v', such that $u' \times m = v' \times n + d$.

From Bezout's Identity to leader election and d-election. Bezout's Identity is the cornerstone on which the proposed algorithm relies when $m \geq 1$. As observed previously, assuming $\gcd(m,n) = d$, there is a pair of positive integers $\langle u, v\rangle$ such that Bezout's Identity can be formulated as $u \times m = v \times n + d$.

Hence, let us consider a rectangle matrix $M[1..m, 1..u]$. Because $\gcd(m,n) = d$, this matrix has $u \times m = v \times n + d$ entries. The basic idea of the algorithm is the following. The n processes compete to own entries of M. After each of them has won exactly v entries, due to the equality $u \times m = v \times n + d$, d entries remain that are not owned by any process. These entries are then used to elect the d leader processes: the winners of these last d competitions become the leaders (leader election corresponds to the case $d = 1$).

To implement these $u \times m = v \times n + d$ competitions, only m anonymous registers are available. So, in addition to its three fields $REG[x].phase$, $REG[x].discarded$, and $REG[x].id$, each register $REG[x]$ has two additional fields $REG[x].level$ and $REG[x].leaders$, such that $0 \leq level \leq u$ and $leaders$ contains a set of processes. When $1 \leq x \leq m$ and $1 \leq level \leq u$, the pair $\langle x, REG[x].level\rangle$ visits all the entries of the (virtual) matrix $M[1..m, 1..u]$. During the last d competitions, the identity of the winner is stored in the field $leaders$ of the corresponding register, providing the processes with a mechanism to retrieve the d global winners.

In order to solve exact d-election when $d = \ell \times \gcd(m,n)$, each register implements $u \times \ell$ competitions (and thus $0 \leq level \leq u \times \ell$). There are then $u \times m \times \ell = v \times n \times \ell + \gcd(m,n) \times \ell$ entries in the matrix $M[1..m, 1..u \times \ell]$. After each process has won $v \times \ell$ competitions, there are exactly $\gcd(m,n) \times \ell = d$ entries left, and thus d leaders are elected. Algorithm 2 solved exact d-election in this general case.

5 BEZOUT-BASED EXACT d-ELECTION FROM $m > 1$ ATOMIC R/W REGISTERS

Algorithm 2 uses the principles outlined in the previous section to extend Algorithm 1 and solve exact d-election. Note that the loop of Algorithm 1 does not contain any subloop or `wait` statement; Algorithm 2 uses this to execute m competitions in parallel by looping over the set of registers, while being a candidate in at most one of these at any given time. At each register, the $u \times \ell$ competitions are then executed sequentially. Globally, processes first solve $v \times n \times \ell$ competitions, and then proceed to elect the d leaders by solving the remaining $\gcd(m,n) \times \ell$ competitions.

This section considers an anonymous memory the size of which is $m \geq 1$ and such that $\gcd(m,n) = d'$, with d being a multiple of d'. The memory is denoted $REG[1..m]$. Due to memory anonymity, $REG[x]$ can denote different registers for distinct processes p_i and p_j. The notation $REG_i[x]$ is then used to denote $REG[x]$ accessed by p_i.

5.1 Local variables

A local copy of the memory.
The local variable $reg_i[1..m]$ is a local copy of $REG_i[1..m]$. It has five fields: $reg_i[x].id$, $reg_i[x].discarded$, $reg_i[x].phase$, $reg_i[x].level$, and $reg_i[x].leaders$.

The fields $phase_i[1..m]$, $discarded_i[1..m]$, and $loser_i[1..m]$ are simple array extensions of their counterparts used in Algorithm 1. The field $reg_i[x].level$ contains the number of competitions that have already been solved at $REG_i[x]$. At the end of the execution, $reg_i[x].leaders$ contains the (global) leaders that have been elected through competitions due to the shared register $REG_i[x]$.

New local variables.

- $level_i[1..m]$ is an array such that $level_i[x]$ contains the number of competitions solved at $REG_i[x]$, as observed by p_i.
- $candidate_i[x]$ is an array of Boolean values, and is such that $candidate_i[x]$ = `true` means that p_i is a candidate in a competition at $REG_i[x]$. It is used to ensure that p_i competes in at most a single competition at a given time: two entries of the array $candidate_i$ cannot be simultaneously `true`.
- to_win_i contains the number of entries of $M[1..m, 1..u \times \ell]$ that p_i still has to win. At the start of the execution, $to\_win_i = v \times \ell$. Once $n \times v \times \ell$ competitions have been solved, to_win_i is set to 1, allowing p_i to compete to be one of the (global) leaders.
- $finishing_i$ is a Boolean initialized to `false`. It is set to the value `true` (line N(3)) when each process has won exactly

```
init: the fields of each REG_i[x] are initialized to:
id ← ⊥, discarded ← ∅, phase ← electing, level ← 0, leaders ← ∅.
Let the constants u and v be the smallest non-negative integers such that
u × m = v × n + gcd(m, n), with d = gcd(m, n) × ℓ.

operation REG_i.collect() is return([REG_i[1], ··· , REG_i[m]]).

operation election(id_i) is % code for process p_i, i ∈ {1, ··· , n}
      phase_i[1..m] ← [electing, ...]; discarded_i[1..m] ← [∅, ...]; loser_i[1..m] ← [false, ...];
      level_i[1..m] ← [0, ..., 0]; candidate_i[1..m] ← [false, ...]: to_win_i ← v × ℓ; finishing_i ← false;
(1)   repeat
(2)     reg_i ← REG_i.collect();
N(1)    solved_i ← 0; for each x ∈ {1, ..., m} do solved_i ← solved_i + reg_i[x].level end for;
N(2)    leaders_i ← ∅; for each x ∈ {1, ..., m} do leaders_i ← leaders_i ∪ reg_i[x].leaders end for;
N(3)    if (solved_i = u × m × ℓ) then return (leaders_i) end if;
N(4)    if ((solved_i ≥ v × n × ℓ) ∧ (finishing_i = false)) then finishing_i ← true; to_win_i ← 1 end if;
N(5)    for each x ∈ {1, ..., m} do
N(6)      if (reg_i[x].level = level_i[x] ∧ reg_i[x].level < u × ℓ)
N(7)        then
(3)           if ((reg_i[x].phase = committing) ∧ (phase_i[x] = electing)) then
(4)                 phase_i[x] ← committing; discarded_i[x] ← ∅
(5)           else if ((|reg_i.discarded| = n − 1) ∧ (reg.id ≠ ⊥))
(6)                 if (reg_i[x].phase = committing)
N(8)                  then if (id_i ∉ reg_i[x].discarded) then % p_i won; progress to next level %
N(9)                          to_win_i ← to_win_i − 1; candidate_i[x] ← false;
N(10)                         if (finishing_i) then reg_i[x].leaders ← reg_i[x].leaders ∪ {id_i} end if;
N(11)                         REG_i[x] ← ⟨⊥, ∅, electing, reg_i[x].level + 1, reg_i[x].leaders⟩
N(12)                 end if
(8)                   else REG_i[x] ← ⟨⊥, ∅, committing, reg_i[x].level, reg_i[x].leaders⟩
(9)                 end if
M(10)         else if ((reg_i[x].id = ⊥) ∧ (¬loser_i[x]) ∧ (∄ y ≠ x : candidate_i[y]) ∧ (to_win_i > 0)) then
N(13)               candidate_i[x] ← true; % p_i cannot compete in various entries at the same time %
(11)                if (reg_i[x].phase = phase_i[x])
(12)                  then discarded_i[x] ← discarded_i[x] ∪ reg_i[x].discarded
(13)                end if;
(14)                REG_i[x] ← ⟨id_i, discarded_i[x], phase_i[x], reg_i[x].level, reg_i[x].leaders⟩
(15)          else if ((reg_i[x].id ∉ {⊥, id_i}) ∧ (id_i ∉ reg_i[x].discarded)) then
N(14)               candidate_i[x] ← false; % p_i lost and can compete elsewhere %
(16)                loser_i[x] ← true; discarded_i[x] ← reg_i[x].discarded ∪ {id_i};
(17)                REG_i[x] ← ⟨⊥, discarded_i[x], phase_i[x], reg_i[x].level, reg_i[x].leaders⟩
(18)          end if
N(15)       else % of the if started at line N(6), hence reg_i[x].level ≠ level_i[x] ∨ reg_i[x].level = u × ℓ %
N(16)         if (reg_i[x].level > level_i[x]) then
N(17)           phase_i[x] ← electing; discarded_i[x] ← ∅; loser_i[x] ← false; level_i[x] ← level_i[x] + 1
N(18)         end if
N(19)     end if  % of the if started at line N(6) %
N(20)   end for  % of the for started at line N(5) %
(19)  end repeat. % of the repeat started at line 1 %.
```

Algorithm 2: Exact d-election for n processes using $m > 1$ anonymous bounded RW registers

$v \times \ell$ entries of the virtual matrix $M[1..m, 1..u \times \ell]$. As $v \times n \times \ell = u \times m \times \ell - d$, it follows that when $finishing_i = \texttt{true}$, exactly d entries of $M[1..m, 1..u \times \ell]$ are not yet owned by a process. The winners of these remaining d entries will be the d processes elected as leaders.

5.2 Description of the algorithm

Extending Algorithm 1 and line identifiers. The design of Algorithm 2 follows a modular approach, namely, this algorithm is incrementally built on top of Algorithm 1. More precisely, the lines identified with the same number describe the same statements in both algorithms (where reg_i is replaced by $reg_i[x]$). The lines prefixed by N are new lines. The line prefixed by M is the corresponding line of Algorithm 1 in which the predicate is extended to take into account the fact there are now $m > 1$ anonymous registers. Expressed differently, if all the lines prefixed by N are suppressed and the last part of the predicate at the line prefixed by M is suppressed, Algorithm 2 boils down to Algorithm 1.

Invocation of election(id_i) *and last competitions predicate.* When a process p_i invokes election(id_i), it first initializes its local variables, and enters a "repeat" loop (lines (1)-(19)) in which it first reads (asynchronously) all the anonymous registers (line (2)). Then, p_i checks if all the entries matrix $M[1..m, 1..u \times \ell]$ have been won by the processes (this is captured by the sum of the levels of all registers, corresponding to the number of competitions solved, being equal to $u \times m \times \ell$, line N(1)). If so, it returns the set of global leaders stored in the fields *leaders* of the registers (lines N(2)-(N3)), and the invocation by p_i terminates. Otherwise, it checks whether $v \times n \times \ell$ entries have been won, and thus only d entries remain. If this is true, p_i assigns the value `true` to $finishing_i$ which will be exploited in the "for" loop that follows (lines N(5)-N(20)).

Global view. After it entered the next iteration of the "repeat" loop, in which it first read the full anonymous memory (line (2)), a

process p_i executes a "for" loop (lines N(2)-N(20)). Each iteration of this loop consists of a local processing associated with a given register $REG_i[x]$ ($1 \leq x \leq m$) of the anonymous memory. The "for" loop can be seen as the execution by p_i of m tasks, each one being on one of the m anonymous registers, all sharing the local variables to_win_i (which counts the entries of $M[1..m, 1..u \times \ell]$ won by p_i), and $finishing_i$ (which allows the processes to terminate).

Hence, when it has executed once the "for" loop, p_i is engaged in a competition (at some level) on each register $REG[x]$. Given a register $REG[x]$, known as $REG_i[y]$ by p_i and $REG_j[z]$ by p_j, etc., the processes p_i, p_j, etc., start participating in the competition for the first entry $M[x, 1]$ (i.e., at level 1), of the virtual matrix M, then the second entry $M[x, 2]$ (i.e., at level 2), etc., i.e., they compete in the same *level* order for each column $M[x, 1..u \times \ell]$. The details of this processing are described below. (Let us notice that, if $m = 1$, the "for" loop involves a single register, and consequently the only loop is then the "repeat" loop.)

Detailed view: inside the "for" loop. (Lines N(2)-N(19)) As just said, a process p_i executes the body of this loop for each register $REG_i[x]$ whose value is locally saved in $reg_i[x]$, $1 \leq x \leq m$. Due to anonymity, in each iteration of the "for" loop, the processes do not necessarily access the registers in the same order, but (as stated above) given a register, they access its levels in the same order. There are two cases.

- Case 1. "Else" part (lines N(15)-N(18)) of the "if" starting at line N(6). If $reg_i[x].level > level_i[x]$, a process has already won the entry $M[x, reg_i[x].level]$, but p_i has not changed of level yet. In this case, it resets its local variables ($phase_i$, $discarded_i[x]$, $loser_i[x]$) to their initial values and jumps to $reg_i[x].level$ to participate in the competition for the matrix entry $M[x, reg_i[x].level]$ (lines (N16)-N(17)).
- Case 2. "Then" part (lines (3)-(18)) of the "if" of line N(3). Like in Algorithm 1, it subdivides into the following four cases.
  - *The second phase of the current competition at $reg_i[x]$ has started.* This is similar to Algorithm 1.
  - *The phase is over.* The predicate $|reg_i.discarded| = n - 1$ is similar to Algorithm 1. If $reg_i[x].phase$ = `committing`, the current competition at $reg_i[x]$ is over. If p_i is the winner, it decreases to_win_i and allows itself to be a candidate in other competitions by setting $candidate_i[x]$ to `false` (line N(9)). If the global leaders are being elected ($finishing_i$ = `true`), p_i is one of them: it then adds (at line N(10)) its identity to $reg_i[x].leaders$. Finally, it increases the level of $reg_i[x]$ and resets the other fields of $reg_i[x]$ (except $reg_i[x].leaders$) in order to allow another competition to start at $reg_i[x]$ (line N(11)).
    If $reg_i[x].phase$ = `electing`, the behavior is similar to Algorithm 1.
  - *p_i competes to be the leader.* There are two differences with Algorithm 1: the use of the local array $candidate_i$ and of the local variable to_win_i. If $\nexists y \neq x : candidate_i[y]$, p_i is not a candidate in a competition at another register; it can then be a candidate at $reg_i[x]$. If $to\_win_i > 0$, p_i still needs to be a candidate in some competitions. This can occur for two reasons: either it has won less than $v \times \ell$ entries (before $finishing_i$ = `true`), or the processes are electing the global leaders (line N(4)), and p_i has not won yet. It then participates as a candidate in the competition at $reg_i[x]$, and sets $candidate_i[x]$ to `true` (line N(13)), thus preventing it from being a candidate elsewhere until it has either won the competition (line N(9) or lost it (line 14).
  - *p_i has lost and must inform the leader.* This is similar to Algorithm 1, except that p_i sets $candidate_i[x]$ to `false`, allowing it to be a candidate in other competitions.

In all these cases, p_i then proceeds to the next anonymous register as defined by the local order processing indicated in its " for" loop. If the "for" loop is terminated (hence p_i visited once more all the anonymous registers), p_i re-enters the "repeat" loop in which it continues to compete, for each $1 \leq x \leq m$, to try to win the matrix entry $M[x, reg_i[x].level]$, and so on until all the competitions are solved, and d global leaders are elected.

5.3 Proof of the Algorithm

Lemma 1. *Assuming all processes invoke* election(), *they all execute the* return() *statement (line N(3)).*

Proof. The loop of Algorithm 1 does not contain any subloop or `wait` statement. Additionally, within the "for" loop, the code of Algorithm 2 is the same as Algorithm 1, except for lines N(8)-N(12) (end of a competition), the condition at line M(10) (p_i can be candidate and still has competitions to win), lines N(13) and N(14) (p_i can only be a candidate in a single competition at once) and lines N(15)-N(19) (level change). Claims 1 and 2 thus hold for each competition for an entry of M (where, in Claim 2, the variable $candidate_i$ translates to $(\neg\exists\, y \neq x : candidate_i[y]) \wedge (to\_win_i > 0)$), and all competitions that have at least one candidate will terminate.

Process p_i initializes to_win_i to $v \times \ell$; until it has won $v \times \ell$ entries, it will then be a candidate in the competition at one of the registers (condition at line M(10)). Processes will then globally solve $n \times v \times \ell$ competitions. At that point, they will all set $finishing_i$ to `true` and set to_win_i to 1 (line N(4)). Exactly d processes will then win remaining competitions, resulting in every register having its level at $u \times \ell$, allowing all the processes to terminate (lines N(1)-N(3)). □ *Lemma 1*

Lemma 2. *Exactly $d = \gcd(m, n) \times \ell$ processes are elected.*

Proof. As stated in Claim 1, once a competition for an entry ends, no other write corresponding to this entry will occur. Once all competitions are solved, there will then be no further modification of the memory, thus all processes will return the same set $leaders_i$ at line N(3).

Once the first $n \times v \times \ell$ competitions have ended and every process p_i sets $finishing_i$ to `true`, the same d competitions are still open for all the processes, even though the registers at which they happen can be addressed differently. A process p_i does not include its identity in any set $reg_i[x].leaders$ before setting $finishing_i$ to `true`, and afterwards, does so when it wins one of the d remaining competitions. Furthermore, p_i can be a candidate in only a single competition after setting $finishing_i$ to `true` (it sets to_win_i to 1 at line N(3). The set $leaders_i$ returned at line N(3) will then contain exactly d different identities, which concludes the proof of the lemma. □ *Lemma 2*

Theorem 4. *Let $d = \gcd(m, n) \times \ell$. Algorithm 2 is a symmetric algorithm that elects exactly d leaders in a system of n asynchronous processes which communicate through m anonymous atomic* RW *registers.*

Proof. The proof follows directly from Lemma 1 and Lemma 2. □ *Theorem 4*

6 TIGHT BOUNDS

Combining Algorithm 2 with the impossibility results proved in Section 2, we have the following corollaries.

Corollary 1. *The condition $\gcd(m, n) = 1$ is necessary and sufficient to elect a (single) leader with a symmetric algorithm in an asynchronous n-process system where communication is through m anonymous atomic RW registers or m anonymous atomic RMW registers.*

Corollary 2. *The condition $\gcd(m, n) \leq d$ is necessary and sufficient to elect at least one and at most d leaders with a symmetric algorithm in an asynchronous n-process system where communication is through m anonymous atomic RW registers or m anonymous atomic RMW registers.*

Corollary 3. *The condition $\gcd(m, n)$ divides d is necessary and sufficient to elect exactly d leaders with a symmetric algorithm in an asynchronous n-process system where communication is through m anonymous atomic RW registers or m anonymous atomic RMW registers.*

Proof. The proof of Corollaries 1, 2 and 3 is an immediate consequence of (i) Theorem 1, and (ii) the very existence of Algorithm 2, that solves exact d-election under the assumption that d is a multiple of $\gcd(m, n)$, using m anonymous atomic RW registers. □ *Corollaries 1, 2 and 3*

7 CONCLUSION

Considering an anonymous shared memory made up of m anonymous RW or RMW registers, this paper focused on the d-election problem in which all the processes participate to elect at least one and at most d processes (the classical leader election problem is 1-election). Exact d-election is the case where exactly d leaders must be elected. It has presented the following results.

- On the impossibility side: proofs showing that
  - there is no symmetric d-election algorithm for $n \geq 2$ processes using m anonymous RMW registers if $\gcd(m, n) > d$, and
  - there is no symmetric exact d-election algorithm for $n \geq 2$ processes using m anonymous RMW registers if $\gcd(m, n)$ does not divide d.
- On the constructive side:
  - an algorithm solving exact d-election when $\gcd(m, n)$ divides d. This algorithm is based on Bezout's identity (a Diophantine equation relating numbers according to their Greatest Common Divisor), and
  - the fact that 1-election can be solved using only one single bounded RW register, which refutes a PODC'1989 conjecture stated in the context of non-anonymous memory systems, claiming that three registers are necessary.

ACKNOWLEDGMENTS

E. Godard, D. Imbs, and M. Raynal were partially supported by the French ANR project DESCARTES (16-CE40-0023-03) devoted to layered and modular structures in distributed computing.

REFERENCES

[1] Z. Aghazadeh, D. Imbs, M. Raynal, G. Taubenfeld, and P. Woelfel. Optimal memory-anonymous symmetric deadlock-free mutual exclusion. In *Proceedings of the 38th ACM Symposium on Principles of Distributed Computing, PODC 2019*, pages 157–166. ACM, 2019.

[2] D. Angluin. Local and global properties in networks of processors (extended abstract). In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing, STOC 1980*, pages 82–93. ACM, 1980.

[3] H. Attiya, A. Gorbach, and S. Moran. Computing in totally anonymous asynchronous shared memory systems. *Information and Computation*, 173(2):162–183, 2002.

[4] F. Bonnet and M. Raynal. Anonymous asynchronous systems: the case of failure detectors. *Distributed Computing*, 26(3):141–158, 2013.

[5] Z. Bouzid, M. Raynal, and P. Sutra. Anonymous obstruction-free (n, k)-set agreement with $n - k + 1$ atomic read/write registers. *Distributed Computing*, 31(2):99–117, 2018.

[6] J. E. Burns and N. A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, 1993.

[7] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[8] E. Godard, D. Imbs, M. Raynal, and G. Taubenfeld. Anonymous read/write memory: Leader election and de-anonymization. In *Proceedings of the 26th International Colloquium on Structural Information and Communication Complexity, SIROCCO 2019*, LNCS 11639, pages 246–261. Springer, 2019.

[9] R. Guerraoui and E. Ruppert. Anonymous and fault-tolerant shared-memory computing. *Distributed Computing*, 20(3):165–177, 2007.

[10] L. Lamport. On interprocess communication. part I: basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[11] S. Navlakha and Z. Bar-Joseph. Algorithms in nature: The convergence of systems biology and computational thinking. *Molecular systems biology*, 7:546, 2011.

[12] S. Navlakha and Z. Bar-Joseph. Distributed information processing in biological and computational systems. *Communications of the ACM*, 58(1):94–102, 2015.

[13] S. Rashid, G. Taubenfeld, and Z. Bar-Joseph. Genome-wide epigenetic modifications as a shared memory consensus problem. *CoRR*, abs/2005.06502, 2020. Also, in the *6th Workshop on Biological Distributed Algorithms, BDA 2018*, London, 2018.

[14] M. Raynal. *Concurrent Programming - Algorithms, Principles, and Foundations.* Springer, 2013. 515 pages, ISBN 978-3-642-32026-2.

[15] M. Raynal. *Distributed Algorithms for Message-Passing Systems.* Springer, 2013. 510 pages, ISBN 978-3-642-38122-5.

[16] M. Raynal. *Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach.* Springer, 2018. 492 pages, ISBN 978-3-319-94140-0.

[17] M. Raynal and G. Taubenfeld. Fully anonymous consensus and set agreement algorithms. In *Proceedings of the 8th International Conference on Networked Systems, NETYS 2020*, LNCS. Springer, June 2020.

[18] M. Raynal and G. Taubenfeld. Mutual exclusion in fully anonymous shared memory systems. *Information Processing Letters*, 158:105938, 2020.

[19] E. Styer and G. L. Peterson. Tight bounds for shared memory symmetric mutual exclusion problems. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing, PODC 1989*, pages 177–191. ACM, 1989.

[20] G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming.* Prentice-Hall, Inc., 2006. 423 pages, ISBN 0-131-97259-6.

[21] G. Taubenfeld. Coordination without prior agreement. In *Proceedings of the 36th ACM Symposium on Principles of Distributed Computing, PODC 2017*, pages 325–334. ACM, 2017.

[22] M. Yamashita and T. Kameda. Computing on anonymous networks: Part i-characterizing the solvable cases. *IEEE Transactions on Parallel and Distributed Systems*, 7(1):69–89, 1996.