Leader-based De-anonymization of an Anonymous Read/Write Memory

Emmanuel Godard[†], Damien Imbs[†], Michel Raynal[⋆,⋄], Gadi Taubenfeld[◦]

[†]LIS, Université d'Aix-Marseille, France
[⋆]Univ Rennes IRISA, Inria, Cnrs, France
[⋄]Department of Computing, Polytechnic University, Hong Kong
[◦]The Interdisciplinary Center, Herzliya, Israel

Abstract

A new notion of anonymity was recently introduced at PODC 2017, namely, anonymity on the names of the registers that define the shared memory. As an example, a shared register named A by a process p and a shared register named B by another process q may correspond to the very same shared register X, while the same name C may correspond to different shared registers for p and q.

Considering an asynchronous n-process anonymous shared memory system, this paper is concerned with the de-anonymization of the memory, i.e., at the end of the execution of a de-anonymization algorithm, the processes must agree on the same name for each shared register, and different shared registers must have different names.

To this end, the paper first addresses leader election in an anonymous memory system. Let n be the number of processes and m the size of the anonymous memory (total number of anonymous registers). It is first shown that there is no election algorithm when the number of anonymous registers is a multiple of n. Then, assuming $m = \alpha\, n+\beta$, where α is a positive integer, three election algorithms are presented, which consider the cases $\beta = 1$, $\beta = n - 1$, and $\beta \in M(n)$, where the set $M(n)$ characterizes the values for which mutual exclusion can be solved despite memory anonymity.

Once election is solved, a general (and simple) de-anonymization algorithm is presented, which takes as a subroutine any memory anonymous leader election algorithm. Hence, any instance of this algorithm works for the values of m required by the selected underlying election algorithm. As the underlying election algorithms, the de-anonymization algorithm is symmetric in the sense that process identities can only be compared for equality.

Keywords: Anonymous read/write registers; Leader election; Memory de-anonymization; Symmetry breaking; Synchronization barrier.

1 Anonymous Memory, Model, and Aim of the Article

1.1 Anonymous Memory

While the notion of *process anonymity* has been studied for a long time from an algorithmic and computability point of view, both in message-passing systems (e.g., [1, 5, 29]) and shared memory systems (e.g., [3, 6, 13]), the notion of *memory anonymity* has been introduced only very recently [27]. (The interested reader will find in [22] an introductory survey on process and memory anonymity.)

Let us consider a shared memory SM made up of m atomic read/write registers [15]. Such a memory can be seen as an array with m entries, namely $SM[1..m]$. In a non-anonymous memory system, for each index x, the name $SM[x]$ denotes the same register whatever the process that invokes the address $SM[x]$. As stated in [27], 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 and synchronization rules the processes have to follow (liveness) to progress without violating the consistency properties associated with the application they solve (safety) [21, 26].

This agreement on the register names no longer exists in a memory-anonymous system. In such a system, while each process knows n (the total number of processes) and m (the size of the memory), the very same identifier $SM[x]$ invoked by a process p_i and invoked by a different process p_j does not necessarily refer to the same atomic read/write register. More precisely, an anonymous memory system is such that:

- prior to the execution, an adversary defined, for each process p_i, a permutation $f_i()$ over the set $\{1, 2, \cdots, m\}$ such that, when p_i uses the address $SM[x]$, it actually accesses $SM[f_i(x)]$,
- no process knows the permutations, and
- all the registers are initialized to the same default value.

Table 1 presents a simple example of a memory made up of three anonymous registers.

As they can have different names for different processes, read/write anonymous registers are necessarily multi-writer/multi-reader registers. The "same initialization" item prevents memory anonymity from being broken with an appropriate register initialization, which could allow different registers to be distinguished, rendering the memory no longer fully anonymous.

identifiers for an external observer	identifiers for process p	identifiers for process q
$SM[1]$	$SM[2]$	$SM[3]$
$SM[2]$	$SM[3]$	$SM[1]$
$SM[3]$	$SM[1]$	$SM[2]$
permutation	$2, 3, 1$	$3, 1, 2$

Table 1: Example of an anonymous memory made up of three registers $SM[1, 2, 3]$

The work described in [27] presents algorithms and impossibility results for mutual exclusion[1], consensus, election and renaming, in an asynchronous read/write anonymous memory system[2]. Among these results, one states a condition on the size m of the anonymous memory which is necessary for any symmetric deadlock-free mutual exclusion algorithm, where *symmetric* means that process identities can

[1] This is the oldest synchronization problem [7]. It consists in ensuring that at most one process at a time can enter some predefined code, called *critical section*. Deadlock-freedom is a progress condition stating that if one or more processes want to enter the critical section, at least one process will succeed.

[2] The consensus and renaming algorithms presented in [27] satisfy the obstruction-freedom progress condition, namely, if a process executes alone during a long enough period, it eventually decides.

only be compared for equality (hence, there is no order notion on process identities). More precisely, given an n-process system where $n \geq 2$, a necessary condition for deadlock-free mutual exclusion algorithm states that the size m must belong to the set $M(n) = \{\ m\ $ such that $\forall\ \ell :\ 1 < \ell \leq n$: $\mathsf{gcd}(\ell, m) = 1\} \setminus \{1\}$. Recently, an anonymous memory deadlock-free mutual exclusion algorithm has been presented in [2] which works for any $m \in M(n)$, thereby showing that this condition is also sufficient for symmetric deadlock-free mutual exclusion in asynchronous read/write anonymous memory systems.

1.2 Computing Model

Processes. The system is composed of a finite set of $n \geq 2$ asynchronous processes denoted p_1, .., p_n. The subscript i in p_i is only a notation convenience, which is not known by the processes. *Asynchronous* means that each process proceeds at its own speed, which can vary with time and always remains unknown to the other processes. Initially, each process p_i only knows its identity id_i, the numbers n and m. No two processes have the same identity. It is assumed that (1) the participation of all the processes is required, and that (2) processes do not fail.

Anonymous shared memory. The shared memory is made up of m atomic unbounded anonymous read/write registers denoted $SM[1...m]$. As a system composed of a single atomic register is not anonymous, it is assumed that $m > 1$. Hence, *all* registers are anonymous. As already indicated, when a process p_i invokes the address $SM[x]$, it actually accesses $SM[f_i(x)]$, where $f_i()$ is a permutation statically defined once and for all by an external adversary.

To stress the fact that the anonymous register $SM[x]$ is not necessarily the same register when used by different processes, we use the notation $SM_i[x]$ when p_i invokes $SM[x]$.

Symmetry constraint on the algorithms. A *symmetric algorithm* is an "algorithm in which the processes are executing exactly the same code and the only way for distinguishing processes is by comparing their identifiers" [27]. Identifiers can be written, read, and compared, but there is no way of looking inside an identifier. Thus, as an example, it is not possible to know whether an identifier is odd or even. Furthermore, the only comparison that can be applied to identifiers is equality. There is no order structuring the identifier name space. (Other notions of symmetry are described in [10, 14]). Let us notice that as all the processes have the same code and all the registers are initialized to the same value, process identities become a key element when one has to design an algorithm in such a constrained context. The reader may notice that this "symmetry" property on process identifiers is the "last step" before process anonymity.

1.3 Problems Addressed in this Article

Leader election. In this problem, the input of each process p_i is its identity id_i. Its output will be deposited in a write-once local variable $leader_i$. The aim is to design an algorithm that provides the local variable $leader_i$ of each process p_i with the same process identity. The only process such that $leader_i = id_i$ is the elected process.

Anonymous memory de-anonymization. In this problem, as before, the input of each process p_i is its identity id_i. The aim is for each process p_i to compute an addressing function $\mathsf{map}_i()$, which is a permutation over the set of the memory indexes $\{1, \cdots, m\}$, such that the two following properties are satisfied.

- Safety. Let $y \in \{1, \cdots, m\}$. For any pair of processes p_i and p_j, and any index $y \in \{1, ..., m\}$, $SM_i[\mathsf{map}_i(y)]$ and $SM_j[\mathsf{map}_j(y)]$ denote the same shared register.

- Liveness. Eventually any process p_i (i) has computed its mapping function $\mathsf{map}_i()$, and (ii) knows that each process has computed its mapping function.

Synchronization barrier. The algorithms presented in this article use the notion of a synchronization barrier. As defined in [26], a synchronization barrier is *a coordination mechanism* that forces the processes that participate in a concurrent program to wait until each of them has reached a certain point in its code. The collection of these coordination points is called the barrier. When a process attains its local barrier point, it has to wait until some predicate (involving shared registers) is satisfied. The fact this predicate becomes satisfied depends on one or more processes (possibly all of them) that have previously assigned appropriate values to shared registers involved in the predicate.

1.4 Motivation and content

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 necessary and sufficient conditions, helps us determine the weakest system assumptions under which fundamental problems, such as leader election can be solved. More generally, the anonymous shared memory model enables to better understand the intrinsic limits for coordinating the actions of asynchronous processes.

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 [19]. Hence, anonymous shared memories could be useful in biologically inspired distributed systems [16, 17]. If this is the case, mastering de-anonymization and leader election in such an adversarial context could reveal to be important from an application point of view.

Content. This article first presents (Section 2) an impossibility result on the values of m for electing a leader or de-anonymizing an anonymous memory. More precisely, if m is a multiple of n, an anonymous memory is not strong enough (from a computability point of view) to break the symmetry constraint on process identifiers.

The article then presents three symmetric anonymous memory election algorithms. Let $m = \alpha\, n+\beta$ (size of the anonymous memory) where α is a positive integer. The three election algorithms differ in the value of β, which constitutes a seed used to break symmetry (defined by the memory anonymity and the symmetric constraint on process identities). They address the following cases.

- $\beta = 1$ (Section 3). The size of the anonymous memory is then $m = \alpha\, n + 1$.
- $\beta = n - 1$ (Section 4). The size of the anonymous memory is then $m = \alpha\, n + (n - 1)$.
- $\beta \in M(n)$ (Section 5) where $M(n) = \{m$ such that $\forall\, \ell :\ 1 < \ell \leq n$: $\mathsf{gcd}(\ell, m) = 1\} \setminus \{1\}$. As already indicated, $M(n)$ is the exact set of values for which deadlock-free mutual exclusion can be solved [2, 27].

Finally, the article presents a general and simple de-anonymization algorithm which uses as subroutine any anonymous memory leader election algorithm (Section 6). This algorithm adds no specific constraint on the value of m. Hence, it works for all the values of m for which the underlying election algorithm it uses is working. Last, Section 8 concludes the article.

We observe that our result for the case where $\beta \in M(n)$ is reminiscent of an interesting results regarding self-stabilizing systems. A self-stabilizing system has the property that it eventually reaches a legitimate configuration when started in any arbitrary configuration. Dijkstra originally introduced the self-stabilization token management problem and gave several solutions for a ring of processes [8]. He also observed that a distinguished process is essential if the number of processes in the ring is *composite* [9]. Later, Burns and Pachl showed that there is a self-stabilizing system with no distinguished

process which solves the problem if the size of the ring is *prime* [4]. The token management problem is essentially equivalent to a variant of a leader election problem where in a legal configuration exactly one process has a special leader flag, and this process remains a leader in any subsequent configuration.

2 An Impossibility Result

Before presenting several algorithms for solving leader election and de-anonymization under various conditions on the number of registers, we present the following impossibility result.[3]

Theorem 1 *There is neither an election algorithm nor a de-anonymizing algorithm, for n processes using m anonymous registers, where $m = \alpha\, n$ and α is a positive integer.*

Proof First, we observe that once de-anonymizing is solved using $m = \alpha\, n$ registers, it is straightforward to solve election using $m = \alpha\, n$ registers. First, run the de-anonymizing algorithm to get $m = \alpha\, n$ non-anonymous registers. Then, using these registers, simply run the symmetric mutual exclusion algorithm from [25] which uses exactly n registers, and let the first process to enter its critical section be the leader. Thus, to prove the theorem, we only need to prove that it is impossible to solve election using $m = \alpha\, n$ registers.

Assume to the contrary, that there is a symmetric election algorithm for n processes using $m = \alpha\, n$ registers where α is a positive integer. Let us arrange the m registers on a ring with m nodes where each register is placed on a different node. We show that the adversary can assign a permutation f_i to each node p_i in such a way that election is not possible.

To each one of the n processes p_i, $1 \leq i \leq n$, let us assign an initial register (namely, the first register that the process accesses) such that, for every two processes p_i and its ring successor p_{i+1} (p_{n+1} is p_1), the distance between their initial registers is exactly α when walking on the ring in a clockwise (increasing modulo) direction. Here we use the assumption that $m = \alpha\, n$.

The lack of global names allows us to assign to 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 $p_1, ..., p_n$)[4], and then let each process take another step, and so on. For process p_i and integer k, let $order(p_i, k)$ denote the k^{th} new register that p_i accesses during an execution where the n processes are running in lock steps, and assume that we arrange that $order(p_i, k)$ 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 $order(p_i, 1)$ is p_i's initial register, $order(p_i, 2)$ is the next new register that p_i accesses and so on. That is, p_i does not access $order(p_i, k+1)$ before accessing $order(p_i, k)$ at least once, but for every $j \leq k$, p_i may access $order(p_i, j)$ several times before accessing $order(p_i, k+1)$ for the first time.[5]

With this arrangement of registers, we run the n processes in lock steps. We have that, for any step k and any process p_i, $SM[order(p_i, k)] = SM[order(p_1, k)]$. Indeed, since only comparisons for equality are allowed, and all registers are initialized to the same value –which (to preserve anonymity) is not a process identity– processes that take the same number of steps will be in the same state, and thus it is not possible to break symmetry. It follows that either all the processes will be elected, or no process will be elected. A contradiction. $\square_{Theorem\ 1}$

[3] Early impossibility results on leader election in networks of anonymous processes can be found in [1].

[4] A step is the execution of an atomic statement by a process [15].

[5] Once a process accesses a register for the first time, say register $SM[x]$, the adversary may map x to any (physical) register that it hasn't accessed yet. However, when it accesses $SM[x]$ again, it must access the same register it has accessed before when referring to x.

3 Anonymous Memory Leader Election When $m = \alpha\ n + 1$

This section presents and proves correct a leader election algorithm when the size of the anonymous memory is $m = \alpha\ n + 1$, for any positive integer α. A process p_i invokes election(id_i). It is assumed that all the processes invoke election(). Such an invocation returns the identity of the elected process.

3.1 Algorithm

Underlying principle. The idea the algorithm relies on is pretty simple. It is the following. First, each process "acquires" α registers (to this end it deposits its identity and an associated tag in these registers). Hence there is a time after which $\alpha\ n$ anonymous registers have been acquired by the n processes. After this occurred, the processes use the single remaining anonymous register ($m - \alpha\ n = 1$) to compete and elect one of them, which will tag appropriately this register, so that the other processes will learn which process was elected.

Wildcard. The values written in an anonymous register are structured as records composed of several fields. In the algorithms, the symbol "-" appearing in the field of a register is a wildcard meaning that the actual value of this field is irrelevant (it can be any value).

Tags. The first field of a record written in an anonymous register is a tag. The possible tags are denoted `start` (initial tag of all registers), `leader` (that allows a process to announce it is leader), `done` (used by a process to announce it is not a leader), and `desa` (used to announce the de-anonymization started).

Local variables. In addition to $leader_i$, each process p_i manages the following local variables: $towrite_i$, $overwritten_i$, $written_i$, which contain sets of memory indexes, $last_i$ which is a memory index, and nb_i which is a non-negative integer. The meaning of these variables will appear clearly in the text of Algorithm 1 ([6].

First part of the algorithm: tagging $\alpha\ n$ registers (lines 1-13). Each anonymous register $SM[x]$ is initialized to $\langle$`start`, $\perp\rangle$, where $\perp$ is the default value, which can be compared (with equality) with any process identity.

When it invokes election(id_i), a process p_i first writes the pair $\langle$`start`, $id_i\rangle$ in the first (from its point of view) α registers, namely, $SM_i[1]$, ...,$SM_i[\alpha]$ (line 3). Then, it waits until all the registers (except one) are tagged `start`, or a register in which it wrote $\langle$`start`, $id_i\rangle$ has been overwritten. There are consequently two cases.

- If registers in which p_i wrote $\langle$`start`, $id_i\rangle$ have been overwritten (the first part of the predicate of line 5 is then satisfied), p_i updates its local variables $overwritten_i$, nb_i, $towrite_i$ and $last_i$, and re-enters the repeat loop, the goal being to have α registers containing $\langle$`start`, $id_i\rangle$.
- If all the registers except one (i.e., exactly $m - 1 = \alpha\ n$ registers) are tagged `start`, p_i exits the loop.

As we will see in the proof, it follows from this collective behavior of the processes that there is a time at which exactly one register still contains its initial value $\langle$`start`, $\perp\rangle$, while for each $j \in \{1, \cdots, n\}$, exactly α registers contain $\langle$`start`, $id_j\rangle$ (this property is named P1' in the algorithm).

[6]All the algorithms are described in pseudocode in which semicolumns are used as a sequentiality operator separating statements.)

```
init: each SM[x] is initialized to ⟨start, ⊥⟩; m = α n + 1.

operation election(id_i) is % code for process p_i, i ∈ {1, ··· , n}
(01)  towrite_i ← {1, ..., α}; overwritten_i ← ∅; written_i ← ∅; last_i ← α;
(02)  repeat
(03)    for each x ∈ towrite_i do SM_i[x] ← ⟨start, id_i⟩ end do;
(04)    written_i ← (written_i \ overwritten_i) ∪ towrite_i;
        % the next line realizes an asynchronous read of the registers SM_i[1], ...,SM_i[n]
        % this line constitutes a predicate controlling a synchronization barrier
(05)    wait until((∃ x ∈ written_i : SM_i[x] ≠ ⟨start, id_i⟩)
                   ∨ (|{ℓ such that SM_i[ℓ] ≠ ⟨start, ⊥⟩}| = α n));
(06)    if (|{ℓ such that SM_i[ℓ] ≠ ⟨start, ⊥⟩}| = α n)
(07)      then exit repeat loop
(08)      else overwritten_i ← { x ∈ written_i such that SM_i[x] ≠ ⟨start, id_i⟩};
(09)           nb_i ← |overwritten_i|;
(10)           towrite_i ← {last_i + 1, ..., last_i + nb_i};
(11)           last_i ← last_i + nb_i
(12)    end if
(13)  end repeat;
      % Property P1': There is a time at which exactly one register contains ⟨start, ⊥⟩
      % and, for each j ∈ {1, ··· , n}, α registers contain ⟨start, id_j⟩
(14)  let ℓ_i be such that SM_i[ℓ_i] = ⟨start, ⊥⟩ or SM_i[ℓ_i] = ⟨leader, −⟩;
(15)  SM_i[ℓ_i] ← ⟨leader, id_i⟩;
      % the last writer in SM_i[ℓ_i] = ⟨start, ⊥⟩ becomes the leader
(16)  wait until((SM_i[ℓ_i] ≠ ⟨leader, id_i⟩)
                 ∨ (SM_i[1..m] has exactly α + 1 entries not tagged done));
(17)  for each x such that SM_i[x] = ⟨start, id_i⟩ do SM_i[x] ← ⟨done, id_i⟩ end for;
      % Property P2': There is a time at which:
      % an index ℓ ∈ {1, ··· , n} is such that a register contains ⟨leader, id_ℓ⟩, and
      % for each j ∈ {1, ··· , n}, there are α registers containing ⟨done, id_j⟩
(18)  if (SM_i[ℓ_i] ≠ ⟨leader, id_i⟩) then
              wait until((SM_i[1..m] has only one entry not tagged done)
                         ∨ (∃ x : SM_i[x] = ⟨desa, −, −⟩))
      % The predicate ∃ x : SM_i[x] = ⟨desa, −, −⟩) can be suppressed if we want only
      % to elect a leader, it implements a synchronization barrier related to de-anonymization
(19)  end if;
(20)  ⟨−, id⟩ ← SM_i[ℓ_i]; leader_i ← id;
(21)  return(leader_i).
```

Algorithm 1: n-process election with $m = \alpha\, n + 1$ anonymous read/write registers

Second part of the algorithm: electing the leader (lines 14-21). As just seen, the previous part of the algorithm has identified a single register of the anonymous memory, namely the only one still containing $\langle \texttt{start}, \bot \rangle$. This register is known by all the processes, more precisely, it is known as $SM_i[\ell_i]$ by p_i, $SM_j[\ell_j]$ by p_j, etc.

So, to become the leader, each process p_i writes the pair $\langle \texttt{leader}, id_i \rangle$ in this register (line 15). It follows that the last process that will write this register will be the leader. There are then two cases.

- If p_i discovers it has not been elected (we have then $SM_i[\ell_i] \neq \langle \texttt{leader}, id_i \rangle$, first predicate of line 16), it resets all the registers containing its tagged identity ($\langle \texttt{start}, id_i \rangle$) to the value $\langle \texttt{done}, id_i \rangle$ (line 17). Then, p_i waits (line 18) until all registers except one are tagged $\langle \texttt{done}, - \rangle$ or a register is tagged `desa`. While p_i is in the wait statement of line 18, due to process asynchrony it is possible that the leader is engaged in the de-anonymization algorithm, in which case it is writing triplets $\langle \texttt{desa}, -, - \rangle$ in each register (see Section 6). If the aim is only to elect a leader,

the second predicate of line 18 can be suppressed.

- If p_i is the last process to write in the register locally known as $SM_i[\ell_i]$, it waits until all the other processes have written $\langle \texttt{done}, -\rangle$ in the registers containing their identity (second part of the predicate of line 15). When this is done, the elected process p_i writes $\langle \texttt{done}, id_i\rangle$ in all the registers containing its identity (line 16), which allows each other process not to remain blocked at line 18.

Finally, p_i being the elected process or not, it writes the identity of the leader in its local variable $leader_i$ (line 20) and returns it (line 21).

As before, we will see in the proof, that there is a time from which there is exactly one index $\ell \in \{1, \cdots, n\}$ such that a register contains $\langle \texttt{leader}, id_\ell\rangle$, and, for each $j \in \{1, \cdots, n\}$, there are α registers containing $\langle \texttt{done}, id_j\rangle$ (This property is named P2 in the algorithm).

3.2 Proof of Algorithm 1

Lemma 1 (Property P1') *Before a process executes line* 14*, there is a finite time at which one register contains* $\langle \texttt{start}, \perp\rangle$*, and, for each* $j \in \{1, \cdots, n\}$*,* α *registers contain* $\langle \texttt{start}, id_j\rangle$*.*

Proof Considering time instants before a process executes line 15, we have the following.

- Let us first observe that the order on the entries of $SM[1..m]$ in which p_i writes them has been statically predefined by the adversary (namely, according to the –unknown– permutation $f_i()$: $SM_i[x]$ is actually $SM[f_i(x)]$). The important point is that a process p_i never backtracks, nor loops on the registers, while scanning $SM[1..m]$, and its successive accesses are $SM[f_i(1)]$, $SM[f_i(2)]$, etc.
- The first write of a process p_i involve the registers $SM_i[1]$ until $SM_i[\alpha]$ (lines 1 and 3). Then, as indicated above, its next writes in SM follows a statically predefined order. The process p_i issues a write of $\langle \texttt{start}, id_i\rangle$ in a register it has not yet written, for each of its previous writes that have been overwritten by another process (line 4). These writes by p_i concern entries of $SM_i[1..n]$ in which it has not yet written (management of the local variables $towrite_i$, $overwritten_i$, $written_i$, and $last_i$, at lines 1, 4, and 8-11). As p_i writes only in new registers, it follows that, for any p_i we have $|\{x \text{ such that } SM[x] = \langle \texttt{start}, id_i\rangle\}| \leq \alpha$, and from a global point of view we have
$$\sum_{i=1}^{n} \big(|\{x \text{ such that } SM[x] = \langle \texttt{start}, id_i\rangle\}|\big) \leq n\ \alpha.$$
- It follows from $m = \alpha\ n + 1$ and the previous inequality, that there is enough room in the array $SM[1..m]$ for each process p_i to write n times the pair $\langle \texttt{start}, id_i\rangle$. Consequently, there is time after which the first predicate of line 5 is false for each process p_i, and as $m = n\alpha + 1$, the remaining entry of $SM[1..m]$ has still its initial value, namely $\langle \texttt{start}, \perp\rangle$, from which we conclude that a process neither remains forever blocked at line 4, nor forever executes the "repeat" loop (lines 2- 13).

It follows from the previous observations that before a process executes line 15, there is a time at which, for each identity id_i, the pair $\langle \texttt{start}, id_i\rangle$ is present in α entries of $SM[1..m]$, and an entry of $SM[1..m]$ has still its initial value, which concludes the proof of the lemma. $\square_{Lemma\ 1}$

The number of write accesses between line 3 and line 13. When considering the proof of Lemma 1, it is easy to count the number of writes in the anonymous memory. In the best case, the (unknown) permutations assigned by the adversary to the processes are such that no process overwrites the pairs written by the other processes. In this case, line 2 generates $\alpha\ n$ writes into the shared memory.

In the worst case, the permutations assigned by the adversary, and the asynchrony among the processes are such that the first α writes of a process are overwritten $(n-1)$ times, the first α writes of another process are overwritten $(n-2)$ times, etc., until the last process whose none of its first α writes are overwritten. In this case, line 2 generates $\alpha\frac{n(n+1)}{2}$ writes into the anonymous shared memory.

Lemma 2 (Property P2') *There is a finite time at which there is $\ell \in \{1,\cdots,n\}$ such that exactly one register contains $\langle \texttt{leader}, id_\ell\rangle$, and, for each $j \in \{1,\cdots,n\}$, there are α registers containing $\langle \texttt{done}, id_j\rangle$.*

Proof It follows from Lemma 1 that no process blocks or loops forever in the "repeat" loop (lines 2-13). Hence, each process eventually executes lines 14-15. Let p_ℓ the last process that executes line 15. This means that after it executed this line, we have $SM_i[\ell_i] = \langle \texttt{leader}, id_\ell\rangle$ for any process p_i (namely, p_ℓ is the process that has been elected). There are two cases.

- A process p_i that is not the leader, is such that $SM_i[\ell_i] \neq \langle \texttt{leader}, id_i\rangle$. Consequently, it cannot be blocked at line 16. So, such a process p_i eventually writes $\langle \texttt{done}, id_i\rangle$ in the α registers containing $\langle \texttt{start}, id_i\rangle$ (line 17). Let us recall that, due to Property $P1'$, these α registers do exist. When the $(n-1)$ processes that are not leader have executed line 17, there are $\alpha(n-1)$ registers containing $\langle \texttt{done}, -\rangle$, α registers containing $\langle \texttt{start}, id_\ell\rangle$, and one register containing $\langle \texttt{leader}, id_\ell\rangle$.
- As far as the leader process p_ℓ is concerned, we have the following. Due to the previous item, the second predicate of line 16 is eventually satisfied. When this occurs, p_ℓ writes $\langle \texttt{done}, id_\ell\rangle$ in the α registers containing $\langle \texttt{start}, id_\ell\rangle$ (line 17) and, from then on, a single register is not tagged $\langle \texttt{done}, -\rangle$, namely the one containing $\langle \texttt{leader}, id_\ell\rangle$.

The lemma follows directly from the two previous items. $\square_{Lemma\ 2}$

Theorem 2 *Algorithm* 1 *solves the election problem in an anonymous memory system made up of $m = \alpha\ n + 1$ registers.*

Proof Once Property P2' is satisfied, no non-leader process is blocked at line 18, and each process eventually executes line 20-21. When this occurs, they all agree on the very same leader, namely the only process p_ℓ whose identity is tagged `leader`. $\square_{Theorem\ 2}$

4 Anonymous Memory Leader Election When $m = \alpha\ n + (n-1)$

Principle of the algorithm. Algorithm 1, which solves the election problem for a system of $m = \alpha\ n + 1$ anonymous registers, was based on the fact that each process can write its identity in α registers that –after some finite time– will not be overwritten, and when this occurred, the single not yet written anonymous register is used to elect the leader.

The principle that underlies the election when there are $m = \alpha\ n + (n-1)$ anonymous registers is dual in the following sense. We have now $m = \alpha\ n + (n-1) = (\alpha+1)(n-1) + \alpha$. So, now each of $(n-1)$ processes write its identity in $\alpha+1$ anonymous registers, while the remaining process can write it in α registers only. When this occurs, the corresponding process becomes elected.

Algorithm. The operational capture of this idea constitutes Algorithm 2, which is obtained from a simple adaptation of Algorithm 1 to the fact that the leader is selected from a memory occupation criterion (instead of competition on a single read/write register, allowing the last writer to be the winner).

init: each $SM[x]$ is initialized to $\langle \texttt{start}, \perp \rangle$; $m = (\alpha\, n + (n-1))$.

operation election(id_i) **is** % code for process p_i, $i \in \{1, \cdots, n\}$
(01) $towrite_i \leftarrow \{1, ..., \alpha + 1\}$; $overwritten_i \leftarrow \emptyset$; $written_i \leftarrow \emptyset$; $last_i \leftarrow \alpha + 1$;
(02) **repeat**
(03) **for each** $x \in towrite_i$ **do** $SM_i[x] \leftarrow \langle \texttt{start}, id_i \rangle$ **end do**;
(04) $written_i \leftarrow (written_i \setminus overwritten_i) \cup towrite_i$;
% the next line realizes an asynchronous read of $SM_i[1..m]$
(05) wait until$\big((\exists\, x \in written_i :\ SM_i[x] \neq \langle \texttt{start}, id_i \rangle)$
$\vee\ (|\{\ell \text{ such that } SM_i[\ell] \neq \langle \texttt{start}, \perp \rangle\}| = m)\big)$;
(06) **if** $\big(|\{\ell \text{ such that } SM_i[\ell] \neq \langle \texttt{start}, \perp \rangle\}| = m\big)$
(07) **then** exit repeat loop
(08) **else** $overwritten_i \leftarrow \{\, x \in written_i$ such that $SM_i[x] \neq \langle \texttt{start}, id_i \rangle\}$;
(09) $nb_i \leftarrow |overwritten_i|$;
(10) $towrite_i \leftarrow \{last_i + 1, ..., \min(last_i + nb_i, m)\}$;
(11) $last_i \leftarrow \min(last_i + nb_i, m)$
(12) **end if**
(13) **end repeat**;
% Property P1": There is a time at which there is a process p_ℓ such that α registers contain the
% pair $\langle \texttt{start}, id_\ell \rangle$, and for each $j \in \{1, \cdots, n\} \setminus \{\ell\}$, $\alpha + 1$ registers contain $\langle \texttt{start}, id_j \rangle$
(14) wait until$\big($there is a process p_ℓ such that
exactly α registers each containing $\langle \texttt{start}, id_\ell \rangle$) or $\langle \texttt{leader}, id_\ell \rangle\big)$;
(15) $leader_i \leftarrow id_\ell$ such that there are exactly α registers each containing $\langle \texttt{start}, id_\ell \rangle$) or $\langle \texttt{leader}, id_\ell \rangle\big)$;
(16) **if** $(leader_i = id_i)$
(17) **then for each** $x \in \{1, ..., m\}$ **do**
if $(SM_i[x] = \langle \texttt{start}, id_\ell \rangle)$ **then** $SM_i[x] \leftarrow \langle \texttt{leader}, id_\ell \rangle)$ **end if end for**;
(18) wait until (the registers not tagged `leader` are tagged `done`)
(19) **else for each** $x \in \{1, ..., m\}$ **do**
if $(SM_i[x] = \langle \texttt{start}, id_i \rangle)$ **then** $SM_i[x] \leftarrow \langle \texttt{done}, id_i \rangle)$ **end if end for**;
(20) wait until$\big($each register contains $\langle \texttt{leader}, id_\ell \rangle$ or $\langle \texttt{done}, - \rangle$ or $\langle \texttt{desa}, -, - \rangle\big)$
% $\langle \texttt{leader}, id_\ell \rangle$ is defined at line 14
(21) **end if**;
% Property P2": There is a time at which:
% an index $\ell \in \{1, \cdots, n\}$ is such that α registers contains $\langle \texttt{leader}, id_\ell \rangle$, and
% for each $j \in \{1, \cdots, n\} \setminus \{\ell\}$, $\alpha + 1$ registers containing $\langle \texttt{done}, id_j \rangle$
(22) return($leader_i$).

Algorithm 2: n-process election for $m = \alpha\, n + (n-1)$ anonymous registers

The main difference lies in the management of the local variables $towrite_i$, $overwritten_i$, $written_i$, $last_i$, and nb_i.

The statements P1" and P2" capture the main properties of the algorithm. P1" states there is a time at which α registers contain the same pair $\langle \texttt{start}, id_\ell \rangle$, and for each $j \in \{1, \cdots, n\} \setminus \{\ell\}$, $\alpha + 1$ registers contain $\langle \texttt{start}, id_j \rangle$. Hence P1" allows each process to know which is the leader. P2" states there is a time at which α registers contain the same pair $\langle \texttt{leader}, id_\ell \rangle$, and the other registers contain the pair $\langle \texttt{done}, - \rangle$. Hence, P2" states that, at the end of the algorithm, each process knows that the leader is known by every process.

As in Algorithm 1, the tag `desa` is required to cope with asynchrony. If the leader starts de-anonymization while the processes have not yet invoked it, the leader may modify the content of the anonymous memory by writing $\langle \texttt{desa}, -, - \rangle$ in registers (see Section 6). Hence, from the election algorithm, the tag `desa` has to be seen as a synonym of `leader` or `done`. The proof of Algorithm 2 is a simple adaptation of the proof of Algorithm 1, and we have the following theorem.

init: each $SM[x]$ is initialized to $\langle \texttt{start}, \perp \rangle$; $m = \alpha\ n + \beta$, $\beta \in M(n)$.

operation election(id_i) **is** % code for process p_i, $i \in \{1, \cdots, n\}$
(01) $towrite_i \leftarrow \{1, ..., \alpha\}$; $overwritten_i \leftarrow \emptyset$; $written_i \leftarrow \emptyset$; $last_i \leftarrow \alpha$;
(02) **repeat**
(03) **for each** $x \in towrite_i$ **do** $SM_i[x] \leftarrow \langle \texttt{start}, id_i \rangle$ **end do**;
(04) $written_i \leftarrow (written_i \setminus overwritten_i) \cup towrite_i$;
(05) wait until$\big((\exists\, x \in written_i :\ SM_i[x] \neq \langle \texttt{start}, id_i \rangle)$
$\vee\ (|\{\ell$ such that $SM_i[\ell] = \langle \texttt{start}, \perp \rangle\}| = \beta)\big)$;
(06) **if** $\big(|\{\ell$ such that $SM_i[\ell] = \langle \texttt{start}, \perp \rangle\}| = \beta\big)$
(07) **then** exit repeat loop
(08) **else** $overwritten_i \leftarrow \{\ x \in written_i$ such that $SM_i[x] \neq \langle \texttt{start}, id_i \rangle\}$;
(09) $nb_i \leftarrow |overwritten_i|$;
(10) $towrite_i \leftarrow \{last_i + 1, ..., last_i + nb_i\}$;
(11) $last_i \leftarrow last_i + nb_i$
(12) **end if**
(13) **end repeat**;
% Property P1"': There is a time at which β registers contain the pair $\langle \texttt{start}, \perp \rangle$,
% and for each $j \in \{1, \cdots, n\}$, α registers contain $\langle \texttt{start}, id_j \rangle$
(14) **let** $betareg_i$ be the set of β indexes $\{d_i^1, d_i^2, ..., d_i^\beta\}$ such that for each $d \in betareg_i$
$SM_i[c] = \langle \texttt{start}, id_i \rangle$ when p_i exited the repeat loop at lines 6-7;
(15) Using the same shared sub-array of (not necessarily contiguous) registers $SM_i[betareg_i]$
the processes execute a symmetric deadlock-free mutex algorithm at the end of which
the last process to enter the critical section is elected. While it is in the critical section,
the elected process p_ℓ write $\langle \texttt{leader}, id_\ell \rangle$ in the β registers of $SM_i[betareg]$
which will allow the other processes to know it is the leader
(16) **if** (p_i is elected) **then** $leader_i \leftarrow id_i$
(17) **else** wait until(β registers are tagged `leader`);
(18) **let** id_ℓ be the id in a register tagged `leader`); $leader_i \leftarrow id_\ell$
(19) **end if**;
(20) **let** $mine_i$ be the set of α indexes $\{c_i^1, c_i^2, ..., c_i^\alpha\}$ such that for each $c \in mine_i$
$SM_i[c] = \langle \texttt{start}, id_i \rangle$ when p_i exited the repeat loop at lines 6-7;
(21) **for each** $x \in mine_i$ **do** $SM_i[x] \leftarrow \langle \texttt{done}, id_i \rangle$ **end for**;
% Property P2"': There is a time at which:
% $\exists\ \ell \in \{1, \cdots, n\}$ is such that β registers contains $\langle \texttt{leader}, id_\ell \rangle$, and
% for each $j \in \{1, \cdots, n\}$, α registers contain $\langle \texttt{done}, id_j \rangle$
(22) wait until(there are $\alpha\ n$ registers such that each of them is tagged `done` or `desa`);
(23) return($leader_i$).

Algorithm 3: Mutex-based election in a system of $m = \alpha\ n + \beta$, $\beta \in M(n)$ anonymous registers

Theorem 3 *Algorithm* 2 *solves the election problem in an anonymous memory system made up of* $m = \alpha\ n + (n - 1)$ *registers.*

5 Anonymous Memory Leader Election When $m = \alpha\ n + \beta$, $\beta \in M(n)$

This section considers the case where an underlying symmetric mutex algorithm, suited to an anonymous memory, is used to elect a leader.

Mutual exclusion in an anonymous memory system. As said in Section 1, mutual exclusion in memory anonymous systems was introduced in [27], which presents a symmetric deadlock-free mutex algorithm for *two* processes only, and a theorem stating that there is no symmetric deadlock-free mutual

exclusion algorithm if the size m does not belong to the set $M(n) = \{\ m$ such that $\forall\ \ell :\ 1 < \ell \leq n$: $\mathsf{gcd}(\ell, m)\ =\ 1\} \setminus \{1\}$. Recently, a symmetric deadlock-free mutual exclusion algorithm has been proposed, which works any number n of processes and for any value $m \in M(n)$ [2], from which follows that $m \in M(n)$ is a necessary and sufficient condition for anonymous mutual exclusion.

Leader election in a system of $m = \alpha\, n + \beta$ anonymous registers, where $\beta \in M(n)$. The idea is to rely on the underlying mutex algorithm to elect a leader. To this end, the processes have first to isolate a set of β anonymous registers in order to be able to execute a symmetric deadlock-free mutex algorithm accessing this subset of registers whose size belongs to $M(n)$.

Algorithm 3 realizes this at lines 1-13, which are a simple adaptation of the same line numbers as in Algorithm 1 and Algorithm 2. When the processes exit the repeat loop (line 13), we have property P1"', namely, there is a time at which β registers contain the pair $\langle \mathtt{start}, \perp \rangle$ and, for each $j \in \{1, \cdots, n\}$ (the set of corresponding indexes is called $betareg_i$ at line 14), and α registers contain $\langle \mathtt{start}, id_j \rangle$ (the set of corresponding indexes is called $mine_i$ at line 20). Hence, the set of β registers define a common anonymous memory, well identified by the n processes, on top of which they can execute a symmetric deadlock-free mutex algorithm (as $\beta \in M(n)$, such mutex algorithms do exist (e.g., [2]). Moreover, as the mutex algorithm is deadlock-free and each process invokes it once, each process eventually enters the critical section. It is shown in [11] how a symmetric deadlock-free mutual exclusion algorithm can be used to allow a process to know it is the last that entered the critical section. The last process to enter is the elected process (lines 15-16).

Finally, when a process p_i is elected, lines 16-21 establish a synchronization barrier such that when a process returns from its invocation of $\mathsf{election}()$, it knows that all processes know the identity of the leader. The proof of Algorithm 3 (left to the reader), is similar to the proof of Algorithm 1 and we have the following theorem.

Theorem 4 *Algorithm* 3 *solves the election problem in an anonymous memory system made up of $m = \alpha\, n + \beta$ registers where $\beta \in M(n) = \{\ m$ such that $\forall\ \ell :\ 1 < \ell \leq n$: $\mathsf{gcd}(\ell, m) = 1\} \setminus \{1\}$.*

Remark. If $\alpha > 1$ and $n + \beta \in M(n)$, instead of requiring the processes to compete for being the leader with a shared memory composed of β anonymous registers, we could have them to compete on a shared memory composed of $(n + \beta)$ anonymous registers, namely the ones that do not contain $\langle \mathtt{start}, \perp \rangle$. As we have then $m = \alpha\ n + \beta = (\alpha - 1)n + (n + \beta)$, each process p_i first writes $\langle \mathtt{start}, id_i \rangle$ in $(\alpha - 1)$ anonymous registers and invokes the mutex algorithm which uses the remaining $n + \beta$ anonymous registers.

6 From Leader Election to De-anonymization

As defined in section 1.3, de-anonymization consists in providing each process with a mapping function $\mathsf{map}()$ such that, given any register index $x \in \{1, ..., m\}$ and any two processes p_i and p_j, $SM_i[\mathsf{map}_i(x)]$ used by p_i and $SM_j[\mathsf{map}_j(x)]$ used by p_j are the same register.

The de-anonymization algorithm is built on top of any election algorithm, so it inherits its constraint on the system parameter pair (n, m). The principle on which it relies it very simple; it consists in directing the elected process p_ℓ to impose its (unknown) permutation $f_\ell()$ to all the processes.

6.1 Leader-based De-anonymization Algorithm: Version 1

The de-anonymization Algorithm 4 is made up of two sequential phases, each terminating with a synchronization barrier.

- Phase 1 (lines 1-7). When p_i invokes de-anonymize(id_i), it first invokes the underlying election algorithm, which returns it the identity of the leader (line 1). There are then two cases according to the fact p_i is the leader or not.
  - If p_i is the leader, for each index $\in 1, ..., m$ it writes $\langle \mathtt{desa}, x, \emptyset \rangle$ in $SM_i[x]$ (line 3). The aim is to impose its local addressing to all the processes. The last field of the triplet is a set initialized to $\emptyset$, that will be used – under the name $KNOWBY$ – at lines 8-10). Hence, the function $\mathsf{map}_i()$ for the leader is $\forall\, y \in \{1, \cdots, m\}$: $\mathsf{map}_i(y) = y$ (line 4).
  - If p_i is not the leader, it waits until the leader has terminated its writes of $\langle \mathtt{desa}, -, \emptyset \rangle$ in all registers (line 5). When this is done, p_i defines its mapping function as $\forall\, y \in \{1, \cdots, m\}$: $\mathsf{map}_i(y) = x$, where $SM_i[x] = \langle \mathtt{desa}, y, - \rangle$ (line 6).

  The previous statements realize a synchronization barrier at the end of which each process p_i knows its mapping function.
- Phase 2 (lines 8-11). The previous synchronization barrier does not allow a process to know that each process knows its mapping function. This is the role of the second phase, which uses the last field of the triplets $\langle \mathtt{desa}, y, - \rangle$.

  When $SM_i[x]$ contains such a triplet, let us call $KNOWBY_i[x]$ its last field. So, when p_i reads $KNOWBY_i[x]$, it reads $SM_i[x]$ and considers only it last field, and when it writes a set set_i in $KNOWBY_i[x]$, it actually writes $\langle \mathtt{desa}, \mathsf{map}_i[x], set_i \rangle$. Let us notice that this does not erase the values of the first two fields of $SM_i[x]$.

  Let us also notice that, due to the mapping function computed by the first phase, the register $SM_i[\mathsf{map}_i(1)]$ is the same for all the processes. This register is now used as a pivot that allows each process to inform the others it knows its mapping function.

  This is done as follows. Each process p_i repeatedly reads $KNOWBY_i[\mathsf{map}_i(1)]$ and adds its identity id_i to $KNOWBY_i[\mathsf{map}_i(1)]$ if and only if id_i is not present in this (initially empty) set (lines 10). Finally, when $KNOWBY_i[\mathsf{map}_i(1)]$ includes the identity of all the processes (line 11), p_i knows that all the processes have computed their mapping function.

Let us observe that, due to process asynchrony, it is possible that, while a process p_i is looping in the repeat loop (lines 9-11), another process p_j returns from its invocation of de-anonymize(id_i), starts executing its upper layer application code on the de-anonymized memory, and writes an application value in $SM_i[\mathsf{map}_i(1)]$. If this occurs, p_i will never terminate, as from its point of view, the value in $SM_i[\mathsf{map}_i(1)]$ is a fake value. This issue is solved by restricting to $SM_i[\mathsf{map}_i(2)..\mathsf{map}_i(m)]$ the part of the de-anonymized shared memory used at the application level. Another way to solve the problem is presented in Section 6.3.

6.2 Proof of the De-anonymization Algorithm Memory

Theorem 5 *If all the processes invoke Algorithm* 4*, they all terminate. Moreover, when the processes have terminated, the following properties are satisfied:*
(i) $\forall\, i, j \in \{1, ...n\}$, $\forall\, y \in \{1, ..., m\}$*:* $SM_i[\mathsf{map}_i(y)]$ *and* $SM_j[\mathsf{map}_j(y)]$ *denote the same register,* (iii) $\forall\, y1, y2 \in \{1, ..., m\}$*:* $(y1 \neq y2) \Rightarrow (SM_i[\mathsf{map}_i(y1)] \neq SM_i[\mathsf{map}_i(y2)])$*, and* (iii) $\forall\, i \in \{1, ...n\}$*, when* p_i *terminates each process has computed its mapping function.*

Proof *On the safety side.* The property (i) follows from the fact that, for any pair of processes p_i and p_j, $SM_i[\mathsf{map}_i(y)]$ and $SM_j[\mathsf{map}_j(y)]$ are the same register, namely $SM_\ell[\mathsf{map}_\ell(y)]$. The property (ii) follows from the fact that (by assumption) the m entries of $SM_\ell[1..m]$ denote different registers. The property (iii) follows from the fact that when $|KNOWBY_i[\mathsf{map}_i(1)]| = n$ is satisfied, all the processes

```
operation de-anonymize(id_i) is
(01) leader_i ← election(id_i);
     % According to the underlying election algorithm, the property P2, or P2" or P2"' is satisfied
(02) if (leader_i = id_i)
(03)   then for each x ∈ {1, ··· , m} do SM_i[x] ← ⟨desa, x, ∅⟩ end for
(04)        let the function map_i() for p_i (leader) be ∀ y ∈ {1, ··· , m}: map_i(y) = y
(05)   else wait until (∀ x ∈ {1, ..., m} : SM_i[x] is tagged desa;
(06)        let the function map_i() for p_i (non-leader)
              be ∀ y ∈ {1, ··· , m}: map_i(y) = x, where SM_i[x] = ⟨desa, y, −⟩
(07) end if;
     % Property D1: here process p_i has computed its mapping function map_i()
     % which is such that for any y ∈ {1, ..., m}: SM_i[map_i(y)] and SM_ℓ[map_ℓ(y)]
     % are the same register, p_ℓ being the elected process
(08) let KNOWBY_i[map_i(1)] denote the third field of SM_i[map_i(1)] (which is a set);
(09) repeat set_i ← KNOWBY_i[map_i(1)];
(10)        if (id_i ∉ set_i) then KNOWBY_i[map_i(1)] ← set_i ∪ {id_i} end if
(11) until |KNOWBY_i[map_i(1)]| = n end repeat;
     % Property D2: when a process passes this synchronization barrier
     % all the processes have computed their mapping function
     % the de-anonymized memory used at upper application layer is SM_i[map_i(2)..map_i(m)]
(12) return().
```

Algorithm 4: Version 1 of the election-based de-anonymization algorithm (code for process p_i)

have deposited their identities in $KNOWBY_i[\mathsf{map}_i(1)]$, which means they all have previously computed their mapping function.

On the liveness side. As the election algorithm terminates (line 1), the leader executes line 3, from which we conclude that all the processes eventually enter the repeat loop. Hence, the proof of the liveness property amounts to show that, eventually each process sees the predicate $|KNOWBY_i[\mathsf{map}_i(1)]| = n$ satisfied. Let us observe that, as it is not provided to the upper layer application, the register $SM_i[\mathsf{map}_i(1)]$ can be modified only at line 9.

Due to line 3 executed by the elected process, $KNOWBY_i[\mathsf{map}_i(1)]$ is initially empty. Then, due to the predicate of line 9, there is a finite time after which any process p_i has written at line 9 a set including its identity id_i in $KNOWBY_i[\mathsf{map}_i(1)]$ (due to the asynchrony, the set it wrote can possibly be overwritten by another process). It follows that there is a finite time τ_1 from which we forever have $|KNOWBY_i[\mathsf{map}_i(1)]| \geq 1$.

Let us consider a process p_i that, after τ_1, reads $KNOWBY_i[\mathsf{map}_i(1)]$ and obtains the set set_i such that $id_i \notin set_i$. It follows from the writing predicate, that p_i writes $set_i \cup \{id_i\}$. As $|set_i| \geq 1$, we have $|set_i \cup \{id_i\}| \geq 2$, and consequently, we have then $|KNOWBY_i[\mathsf{map}_i(1)]| \geq 2$ after the write of p_i. As the processes repeatedly execute the repeat loop, it follows that there is a time τ_2 from which we always have $|KNOWBY_i[\mathsf{map}_i(1)]| \geq 2$.

Using the same reasoning, there is a time τ_3 from which the predicate $|KNOWBY_i[\mathsf{map}_i(1)]| \geq 3$ is always satisfied, etc., until we have $|KNOWBY_i[\mathsf{map}_i(1)]| = n$. ([7]) $\square_{Theorem\ 5}$

[7] Let us notice that the reasoning is based only on the fact that the size of $KNOWBY_i[\mathsf{map}_i(1)]$ increases. It is possible that the set including at least x processes read by a process p_i and the set including at least x processes read by a process p_j are not the same. The important point is that eventually the size of the set $KNOWBY_i[\mathsf{map}_i(1)]$ increases to a value greater than x.

6.3 Leader-based De-anonymization Algorithm: Version 2

This section shows that it is possible to offer the full de-anonymized memory $SM_i[\mathsf{map}_i(1)]..\mathsf{map}_i(m)]$ to the upper layer application, at the price of a control bit permanently contained by each register. This is realized by Algorithm 5 (which extends Algorithm 4) with an additional synchronization barrier.

Each register $SM_i[x]$ is now composed of two fields, one denoted $SM_i[x].BIT$ containing a bit (initialized to 0), and a second field containing the same values as before. The notation used to read/write access $SM_i[x].BIT$ is the same as the one used for $KNOWBY$ in Algorithm 4. The invocation of de-anonymize(id_i) does not modify these bits, which are set to the value 1 by the leader only (let us recall that $leader_i$ is local variable in which p_i saved the identity of the leader).

```
operation de-anonymize2(id_i) is
      de-anonymize(id_i);
(13)  if (leader_i = id_i)
(14)     then for each x ∈ {1, ··· , m} do BIT_i[x] ← 1 end for
(15)     else wait until (∀ x ∈ {1, ...m : BIT_i[x] = 1)
(16)  end if;
      % after this third synchronization barrier, the permanent
      % control information is reduced to a single bit
(17)  return().
```

Algorithm 5: Version 2 of the election-based de-anonymization algorithm (code for process p_i)

As the leader only can modify the additional bits, it follows that, whatever the process asynchrony pattern, the setting of the bits to the value 1 moves the global information "$|KNOWBY_i[\mathsf{map}_i(1)]| = n$" (namely, de-anonymization is over and known by each process)[8] in the bit of each register. Hence, after a process exits line 15, it knows it.

7 Related work

The work on anonymous objects was inspired by Michael O. Rabin's paper on solving the Choice Coordination Problem with k alternatives (k-CCP) [20]. In the k-CCP, n processes must choose between k alternatives. The agreement on a single choice is complicated by the fact that there is *no a priori* agreement on names for the alternatives. That is, each process has its own naming convention for the alternatives. Rabin has assumed that processes communicate by applying RMW operations to exactly k registers which do not have global names. The k different registers represent the k possible alternatives.

In [27], the notion of anonymous objects was defined, and several results were presented for a model where communication is only via anonymous (read/write) registers. The problems addressed were symmetric deadlock-free and obstruction-free mutual exclusion, consensus, election and renaming, for which algorithms and impossibility results were presented. Among the results presented in [27], one states a condition on the size m of the anonymous memory which is necessary for any symmetric deadlock-free mutual exclusion algorithm.

In [2], tight space bounds for solving the symmetric deadlock-free mutual exclusion problem using anonymous read/write registers and anonymous RMW registers, are presented. In particular, anonymous memory deadlock-free mutual exclusion algorithm has been presented in [2] which works for any $m \in M(n)$, thereby showing that the necessary condition from [27], is also sufficient for symmetric deadlock-free mutual exclusion in read/write anonymous memory systems. The open problem from [27], regarding the existence of an anonymous memory two-process starvation-free mutual exclusion algorithm is still open.

[8] This is the information saved in $SM_i[\mathsf{map}_i(1)]$, which can be possibly destroyed as explained at the end of Section 6.1.

It was also shown in [27] that for a model where the number of processes is *not* a priori known (or is unbounded) anonymous registers are strictly weaker than non-anonymous registers. However, when the number of processes is not a priori known, it seems that anonymous registers are *trivial* objects – they cannot be used to solve any problem that requires communication. The question of whether anonymous registers are weaker than non-anonymous registers when the number of processes (participating and non-participating) is known is open.

Results regarding the computational power of anonymous and non-anonymous objects can be found in [28]. In particular, it is proved in [28] that anonymous bits are non-trivial objects which are strictly weaker than anonymous (and hence also non-anonymous) multi-valued registers.

Leader election in read/write anonymous memory systems has been recently addressed in [11, 12]. The present article actually merges and extends these two papers.

Anonymous shared memory systems appear to be useful in modeling biologically inspired distributed computing methods, especially those that are based on ideas from molecular biology [16, 17, 19]. It is shown in [19] how the process of genome wide epigenetic modifications, which allows cells to utilize the DNA, can be modeled as an anonymous shared memory system where, in addition to the shared memory, also the processes (that is, proteins modifiers) are anonymous. Epigenetic refers in part to post-translational modifications of the histone proteins on which the DNA is wrapped. Such modifications play an important role in the regulation of gene expression.

Finally, fully anonymous shared memory systems, where *both* processes and memory are anonymous, were recently investigated in [23, 24].

8 Conclusion

This article is on synchronization problems in an n-process system in which the communication is through m anonymous read/write registers only. In such a system there is no a priori agreement on the names of the registers: the same register name A used by several processes can head them to different registers. In such a context, the article addressed the following problems: leader election and memory de-anonymization. It was first shown that these problems are impossible to solve if $m = \alpha\, n$, where α is a positive integer. Then, considering $m = \alpha\, n + \beta$, it has presented election algorithms for $\beta = 1$, $\beta = n - 1$, and $\beta \in M(n)$ where $M(n)$ is the set of the anonymous memory sizes for which symmetric deadlock-free mutual exclusion can be solved in n-process systems. This is summarized in Table 2, which may help users to favor a leader election algorithm according to the values of m (size of the anonymous memory) and n (number of processes) of the underlying anonymous memory system.

Relation on the pair $\langle n, m\rangle$	Result	Where
$m = \alpha\, n$	Impossible	Section 2, Theorem 1
$m = \alpha\, n + 1$	Algorithm 1	Section 3, Theorem 2
$m = \alpha\, n + (n - 1)$	Algorithm 2	Section 4, Theorem 3
$m \neq 1$ such that $\forall\, \ell :\ 1 < \ell \leq n$: $\mathsf{gcd}(\ell, m) = 1$	Algorithm 3	Section 5, Theorem 4

Table 2: Impossibility/possibility for leader election

Two leader election-based de-anonymization algorithms have also been presented. They differ in the size of the de-anonymized memory they provide the application layer. The first provides a de-anonymized memory in which each register must forever contain a bit indicating it has de-anonymized. The second algorithm provides a de-anonymized memory in which all the bits of the de-anonymized registers can be be used by the application. To this end this algorithm relies on a register which cannot be used by the upper layer application. To summarize the first algorithm provides the n processes with

m de-anonymized registers in each of which a bit cannot be used by the upper layer application, while the second provides the n processes with $(m-1)$ de-anonymized registers in which all the bits can be used.

We point out that once de-anonymization is obtained, it becomes possible to use, for example, a symmetric starvation-free mutex algorithm that uses non-anonymous registers, thereby obtaining a symmetric starvation-free mutex algorithm working on top of an anonymous memory[9]. We emphasize that running a starvation-free mutex algorithm on top of a de-anonymization layer does not solve the original open problem from [27], regarding the existence of a anonymous memory two-process starvation-free mutex algorithm. In the definition of the mutex problem participation is not required (a process may never leave its remainder region), while our implementation of the de-anonymization layer, assumes that participation is required.

As stated in [27], the anonymous memory communication model "enables us to better understand the intrinsic limits for coordinating the actions of asynchronous processes". It consequently enriches our knowledge of what can be (or cannot be) done when an adversary replaced a common addressing function, by individual and independent addressing functions, one per process.

Acknowledgments

The authors want to thank the referees for their precise and constructive comments, which helped improve the presentation. This work was partially supported by the French ANR project DESCARTES (16-CE40-0023-03) devoted to layered and modular structures in distributed computing.

References

[1] Angluin D., Local and global properties in networks of processes. *Proc. 12th Symposium on Theory of Computing (STOC'80)*, ACM Press, pp. 82-93, (1980)

[2] Aghazadeh Z., Imbs D., Raynal M., Taubenfeld G., and Woelfel Ph., Optimal memory-anonymous symmetric deadlock-free mutual exclusion. *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC'19)*, ACM press, pp. 157-166 (2019)

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

[4] Burns J. E. and Pachl J. K., Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11(2):330-344 (1989)

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

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

[7] Dijkstra E.W., Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569 (1965)

[8] Dijkstra E.W., Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643-644 (1974)

[9] Peterson's mutual exclusion algorithm is such a symmetric algorithm [18].

[9] E.W. Dijkstra. Self-stabilization in spite of distributed control (DEW391). *Reprinted in Selected Writing on Computing: A Personal Perspective*, Springer, pp. 41-46 (1982)

[10] Garg V. K. and Ghosh J., Symmetry in spite of hierarchy. *Proc. 10th Int'l Conference on Distributed Computing Systems (ICDCS'90)*, IEEE Computer Press, pp. 4-11 (1990)

[11] Godard E., Imbs D., Raynal M., Taubenfeld G., Mutex-based de-anonymization of an anonymous read/write memory. *Proc. 7th Int'l Conference on Networked Systems (NETYS'19)*, Springer LNCS, 15 pages (2019)

[12] Godard E., Imbs D., Raynal M., Taubenfeld G., Anonymous read/write memory: leader election and de-anonymization. *Proc. 26th Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO'19)*, Springer LNCS 11639, pp. 246-261 (2019)

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

[14] Johnson R. E., and Schneider F. B., Symmetry and similarity in distributed systems. *Proc.4th ACM Symposium on Principles of Distributed Computing (PODC'85)*, pp. 13-22, ACM Press (1985)

[15] Lamport L., On interprocess communication, Part I: basic formalism. *Distributed Computing*, 1(2):77-85 (1986)

[16] Navlakha S. and Bar-Joseph Z., Algorithms in nature: the convergence of systems biology and computational thinking. *Molecular systems biology*, 7(546):1-11 (2011)

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

[18] Peterson G.L., Myths about the mutual exclusion problem, *Information Processing Letters*, 12(3):115-116 (1981)

[19] Rashid S., Taubenfeld G., and Bar-Joseph Z., Genome wide epigenetic modifications as a shared memory consensus problem. *arXiv:2005.06502* (May 2020). Also, in the *6th Workshop on Biological Distributed Algorithms (BDA'18)*, London (2018)

[20] M. O. Rabin. The choice coordination problem. *Acta Informatica*, 17:121–134, 1982.

[21] Raynal M., *Concurrent programming: algorithms, principles and foundations*. Springer, 515 pages, ISBN 978-3-642-32026-2 (2013)

[22] Raynal M. and Cao J., Anonymity in distributed read/write systems: an introductory survey. *Proc. 6th Int'l Conference on Networked Systems (NETYS'18)*, Springer LNCS 11028, pp. 122-140 (2018)

[23] Raynal M. and Taubenfeld G., Mutual exclusion in fully anonymous shared memory systems. *Information Processing Letters*, Volume 158 (June 2020)

[24] Raynal M. and Taubenfeld G., Fully Anonymous Consensus and Set Agreement Algorithms. Proceedings of the 8th international conference on networked systems (NETYS 2020), Morocco, June 2020.

[25] Styer E., and Peterson G. L. Tight bounds for shared memory symmetric mutual exclusion problems. *Proc. 8th ACM Symposium on Principles of Distributed Computing*, ACM Press, pp. 177-191 (1989)

[26] Taubenfeld G., *Synchronization algorithms and concurrent programming*. Pearson Education/Prentice Hall, 423 pages, ISBN 0-131-97259-6 (2006)

[27] Taubenfeld G., Coordination without prior agreement. *Proc. 36th ACM Symposium on Principles of Distributed Computing (PODC'17)*, ACM Press, pp. 325-334 (2017)

[28] Taubenfeld G. Set agreement power is not a precise characterization for oblivious deterministic anonymous objects *Proc. 26th International Colloquium on Structural Information and Communication Complexity (SIROCCO'19)*, Springer LNCS, pp. 293-308 (2019)

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