Anonymous Read/Write Memory: Leader Election and De-anonymization

Emmanuel Godard, Damien Imbs, Michel Raynal, Gadi Taubenfeld

Anonymous Read/Write Memory: Leader Election and De-anonymization

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

[†]LIS, Université d'Aix-Marseille, France
[⋆]Univ Rennes IRISA, France
[‡]Department of Computing, Polytechnic University, Hong Kong
[◦]The Interdisciplinary Center, Herzliya 46150, Israel

Abstract. Anonymity has mostly been studied in the context where processes have no identity. A new notion of anonymity was recently introduced at PODC 2017, namely, this notion considers that the processes have distinct identities but disagree on the names of the read/write registers that define the shared memory. As an example, a register named A by a process p and a shared register named B by another process q may correspond to the very same register X, while the same name C may correspond to different registers for p and q.
Recently, a memory-anonymous deadlock-free mutual exclusion algorithm has been proposed by some of the authors. This article addresses two different problems, namely election and memory de-anonymization. Election consists of electing a single process as a leader that is known by every process. Considering the shared memory as an array of atomic read/write registers $SM[1..m]$, memory de-anonymization consists in providing each process p_i with a mapping function $\mathsf{map}_i()$ such that, for any two processes p_i and p_j and any integer $x \in [1..m]$, $\mathsf{map}_i(x)$ and $\mathsf{map}_j(x)$ allow them to address the same register.
Let n be the number of processes and α a positive integer. The article presents election and de-anonymization algorithms for $m = \alpha\ n + \beta$ registers, where β is equal to 1, $n - 1$, or belongs to a set denoted $M(n)$ (which characterizes the values for which mutual exclusion can be solved despite anonymity). The de-anonymization algorithms are based on the use of election algorithms. The article also shows that the size of the permanent control information that, due to de-anonymization, a register must save forever, can be reduced to a single bit.

Keywords: Anonymous registers, Asynchronous system, Atomic read/write registers, Concurrent algorithm, Leader election, Local memory, Mapping, Memory de-anonymization, Mutual exclusion, Synchronization.

1 Anonymous Memory, Model, and Aim of the Article

1.1 Anonymous Memory

Memory anonymity. 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., [1,4,17]) and shared memory systems (e.g., [3,5,8]), the notion of *memory anonymity* has been introduced only very recently by [15]. (See also [11] for an introductory survey on process and memory anonymity.)

Let us consider a shared memory SM made up of m atomic read/write registers. 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 [15], 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 [10,14].

This a priori agreement does no longer exist in a memory-anonymous system. In such a system 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, a memory-anonymous system is such that:

- prior 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 $SM[f_i(x)]$, and
- no process knows the permutations.

The read/write registers of a memory-anonymous system are necessarily MWMR.

Results on memory anonymity in mutual exclusion. The work described in [15] on anonymous read/write memory addressed mutual exclusion, consensus, election and renaming, problems for which it presented algorithms and impossibility results. The consensus, election and renaming algorithms in [15] satisfy the starvation-freedom progress condition, namely, if a process executes alone during a long enough period, it eventually decides. This progress condition is different from the one considered in this article.

Among the results from [15], one states a condition on the size m of the anonymous memory which is necessary for any symmetric deadlock-free algorithm, where *symmetric* means that process identities can only be compared with equality (hence, there is no notion of a total order on process identities). More precisely, given an n-process system where $n \geq 2$, there is no 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, it has been shown in [2] that the condition $m \in M(n)$ is also a sufficient condition for symmetric deadlock-free mutual exclusion in 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 to its own speed, which can vary with time and remains always unknown to the other processes. Initially, each process p_i knows only its identity id_i, the total number of processes n, and the fact that no two processes have the same identity. It is assumed that there are no process failures. Furthermore, unlike the mutual exclusion model where a process may never leave its remainder region, it is assumed that all the processes must participate in the algorithm.

Anonymous shared memory. The shared memory is made up of m atomic 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. We will use the notation $SM_i[x]$ to denote $SM[f_i(x)]$, to stress the fact that 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 have been possible to distinguish different registers, which consequently will no longer be fully anonymous.

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 identifiers. Identifiers can be written, read, and compared, but there is no way of looking inside an identifier. Thus it is not possible to know whether an identifier is odd or even" [15]. 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 [6,9]). 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.

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 process p_i: $SM_i[\mathsf{map}_i(y)] = SM[y]$.
- Liveness. There is a finite time after which all the processes have computed their addressing function $\mathsf{map}_i()$.

The safety property states that once a process p_i has computed $\mathsf{map}_i()$, its local anonymous memory address $SM_i[x]$, where $x = \mathsf{map}_i(y)$, denotes the shared register $SM[y]$.

1.4 Content

This article presents first an impossibility result. Then, it presents symmetric algorithms solving the two previous problems in a system where the process cooperate through m atomic anonymous read/write registers. As already indicated, it is assumed that all the processes participate in the algorithms, and the size of the memory is $m = \alpha\ n + \beta$, where α is a positive integer and β can take the following values:

- $\beta = 1$. The size of the anonymous memory is then $m = \alpha\, n + 1$.
- $\beta = n - 1$. The size of the anonymous memory is then $m = \alpha\, n + (n - 1)$.
- $\beta \in M(n)$ where $M(n)$ is as defined above. Namely, $M(n)$ is the set of values for which deadlock-free mutual exclusion can be solved [2,15]. This is due to the fact that when $\beta \in M(n)$, the algorithms use a deadlock-free mutual exclusion algorithm to solve conflicts -which do not exist when $\beta = 1$ or $\beta = n - 1$). In this specific case, α can also be 0.

Find a characterization of the set of the values of m for which leader election can be solved in a memory anonymous system remains an open problem (see the Conclusion section).

2 An Impossibility Result

Theorem 1. *There is neither a de-anonymizing algorithm nor an election 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 [13] 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. Let us call the n processes $p_0, ..., p_{n-1}$. To each one of the n processes, we assign an initial register (namely, the first register that the process accesses) such that for every two processes p_i and $p_{i+1\ (mod\ n)}$, the distance between their initial registers is exactly α when walking on the ring in a clockwise direction. Here we use the assumption that $m = \alpha\, n$.

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 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.[1]

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

With this arrangement of registers, we run the n processes in lock steps. 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 at 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 Memory Anonymous Leader Election when $m = \alpha\, n + 1$

3.1 Algorithm

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.

First part of the algorithm: lines 1-12. Each anonymous register $SM[x]$ is initialized to $\langle \mathtt{start}, \perp \rangle$, where $\perp$ is 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 $\langle \mathtt{start}, id_i \rangle$ 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 time at which exactly one register still contains its initial value $\langle \mathtt{start}, \perp \rangle$, while for each $j \in \{1, \cdots, n\}$, exactly α registers contain $\langle \mathtt{start}, id_j \rangle$ (this property is named P1 in the algorithm).

Second part of the algorithm: lines 13-18. As just seen, the previous part of the algorithm has identified a single register of the anonymous memory, namely the only one containing $\langle \mathtt{start}, \perp \rangle$. 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 $\langle \mathtt{leader}, id_i \rangle$ in this register (known as $SM_i[\ell_i]$ by p_i, line 14). 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 $SM_i[\ell_i] \neq \langle \mathtt{leader}, id_i \rangle$, first predicate of line 15), it resets all the registers containing its tagged identity ($\langle \mathtt{start}, id_i \rangle$) to the value $\langle \mathtt{done}, id_i \rangle$ (line 16). Then, p_i waits until all registers except one are tagged $\langle \mathtt{done}, - \rangle$.

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

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$\big((\exists\, x \in written_i :\ SM_i[x] \neq \langle \texttt{start}, id_i \rangle)$
$\vee\ (|\{\ell \text{ such that } SM_i[\ell] \neq \langle \texttt{start}, \perp \rangle\}| = \alpha\, n)\big)$;
(06) **if** $\big(|\{\ell \text{ such that } SM_i[\ell] \neq \langle \texttt{start}, \perp \rangle\}| = \alpha\, n\big)$
(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) $towrite_i \leftarrow \{last_i + 1, ..., last_i + nb_i\}$; $last_i \leftarrow last_i + nb_i$;
(11) **end if**
(12) **end repeat**;
% Property P1: There is a time at which exactly one register contains $\langle \texttt{start}, \perp \rangle$
% and, for each $j \in \{1, \cdots, n\}$, α registers contain $\langle \texttt{start}, id_j \rangle$
(13) **let** ℓ_i **be such that** $SM_i[\ell_i] = \langle \texttt{start}, \perp \rangle$ or $SM_i[\ell_i] = \langle \texttt{leader}, - \rangle$;
(14) $SM_i[\ell_i] \leftarrow \langle \texttt{leader}, id_i \rangle$;
(15) wait until$\big((SM_i[\ell_i] \neq \langle \texttt{leader}, id_i \rangle)$
$\vee\ (SM_i[1..m]$ has exactly $\alpha + 1$ entries not tagged done$)\big)$;
(16) **for each** x such that $SM_i[x] = \langle \texttt{start}, id_i \rangle$ **do** $SM_i[x] \leftarrow \langle \texttt{done}, id_i \rangle$ **end for**;
% Property P2: There is a time from which there is exactly 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 $\langle \texttt{done}, id_j \rangle$
(17) **if** $(SM_i[\ell_i] \neq \langle \texttt{leader}, id_i \rangle)$ **then**
wait until$\big(SM_i[1..m]$ has only one entry not tagged done$\big)$ **end if**;
(18) $\langle -, id \rangle \leftarrow SM_i[\ell_i]$; $leader_i \leftarrow id$.
% Here, one register is tagged leader, all the others are tagged done.

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

– If p_i is the last process to write in the single 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 17 and progress to the last line of the algorithm. When this occurs, each process can assign the identity of the leader to its local variable $leader_i$ (line 18).

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 14, 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 while scanning $SM[1..m]$, and its successive accesses are $SM[f_i(1)]$, $SM[f_i(2)]$, etc.
- The first writes 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 8-10). 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- 12).

It follows from the previous observations that before a process executes line 14, 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 12. 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 a 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 from 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 2-12). Hence, each process eventually executes lines 13-14. Let p_ℓ the last process that executes line 14. 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 15. 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 16). Let us recall that, due to Property $P1$, these exactly α registers do exist. When the $(n-1)$ processes that are not leader have executed line 16, 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 15 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 16) 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.*

Proof Once Property P2 is satisfied, no non-leader process is blocked at line 17, and each process eventually execute line 18. 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 From Leader Election to De-anonymization when $m = \alpha\, n + 1$

4.1 A Simple Leader-based De-anonymization Algorithm

As soon as a process has been elected, it is easy to de-anonymize the anonymous memory. To this end, the elected process p_ℓ imposes its mapping function to all the processes.

Algorithm 2 is such a de-anonymization algorithm, which relies on Property P2. Each process p_i invokes the operation $\mathsf{election}(id_i)$ (line 1). Then for each register $SM_\ell[x]$, the elected process p_ℓ writes the pair $\langle \texttt{desa}, x \rangle$ in $SM_\ell[x]$ (line 3). Hence, its mapping function is $\forall\ x \in \{1, \cdots, m\}$: $\mathsf{map}_i(x) = x$. On the other side, any non-leader process p_i waits until all the registers are tagged `desa` (line 4). When this occurs, p_i computes its own mapping function (line 5), which is such that $\mathsf{map}_i(y) = x$, where $SM_i[x] = \langle \texttt{desa}, y \rangle$. The proof of this algorithm is easy and left to the reader.

As a simple example see Fig. 1, where p_ℓ has been elected as leader, and $f_\ell()$ is the permutation defined by the adversary for p_ℓ (this permutation remains always unknown

operation SM_i.scan() **returns** $([SM_i[1], \cdots, SM_i[m]])$.

operation de-anonymize(id_i) **is**
(01) election(id_i);
% in the following ℓ_i has the value computed in election(id_i); moreover, if p_i is the
% first process that exits from election(id_i):
% one register is tagged `leader`, all the others are tagged `done`
(02) **if** $(SM_i[\ell_i] = \langle$`leader`$, id_i\rangle)$ % this predicate is equivalent to $leader_i = id_i$
(03) **then for each** $x \in \{1, \cdots, m\}$ **do** $SM_i[x] \leftarrow \langle$`desa`$, x\rangle$ **end for**
% the permutation for p_i is: $\forall\, y \in \{1, \cdots, m\}$: $\mathsf{map}_i(y) = y$ %
(04) **else repeat** $sm_i \leftarrow SM_i$.scan() **until** $(\forall\, x :\ sm_i[x]$ is tagged `desa`$)$ **end repeat**;
(05) **for each** $x \in \{1, \cdots, m\}$ **do** $\mathsf{map}_i(y) \leftarrow x$ where $sm_i[x] = \langle$`desa`$, y\rangle$ **end for**
% perm. of p_i is: $\forall\, y \in \{1, \cdots, m\}$: $\mathsf{map}_i(y) = x$, where $sm_i[x] = \langle$`desa`$, y\rangle$
(06) **end if**. % Here, each register $SM_i[x]$ is tagged `desa`.

Algorithm 2: Election-based de-anonymization (code for p_i, $m = \alpha\ n + 1$)

to the processes). $SM_i[x] = \langle$`desa`$, y\rangle$, and $SM_j[z] = \langle$`desa`$, y\rangle$ address the same register, which is $SM_\ell[y]$. Hence, this register is locally known as $SM_i[\mathsf{map}_i(y)]$ by p_i, $SM_j[\mathsf{map}_j(y)]$ by p_j, and $SM_\ell[\mathsf{map}_\ell(y)] = SM_\ell[y]$ by p_ℓ.

4.2 Using the De-anonymized Memory

When a process p_i returns from Algorithm 2, it knows that all the processes will share the same index for the same register (i.e., if $SM_i[x] = \langle$`desa`$, y\rangle$, then $SM_i[\mathsf{map}_i(y)]$ is $SM_i[x]$). When this occurs, process p_i could start executing its local algorithm defined by the upper layer application, but if it writes an application-related value in some of these registers, this value can overwrite a pair tagged `desa` stored in a register not yet read by other processes. A way to prevent this problem from occurring consists in tagging all the values written by a process at the application level by the tag `apply`, and include a field containing the common index y associated with this register. Hence, at the application level, a register will contain $\langle$`apply`$(y), v\rangle$. In this way, despite asynchrony, any process p_j will be able to compute its local mapping function $\mathsf{map}_j()$, and start its upper layer application part, as soon as it has computed $\mathsf{map}_j()$.

Let us notice that one bit is needed to distinguish the tag `desa` and the tag `apply`. Hence each of a pair $\langle$`desa`$, y\rangle$ and a pair $\langle$`apply`$(y), -\rangle$ requires $(1 + \log_2\ m)$ control bits.

4.3 Reducing the Size of the Permanent Control Information

Aim and additional assumption. This section shows that, at the price of an additional synchronization phase, the control information that each register must forever contain can be reduced from $(1 + \log_2\ m)$ to a single bit.

To this end, we assume now that each atomic read/write register $SM[x]$ is composed of two parts $SM[x].BIT$ and $SM[x].RM$ (i.e., $SM[x] = \langle SM[x].BIT,\ SM[x].RM\rangle$). $SM[x].BIT$ is for example the leftmost bit of $SM[x]$, and $SM[x].RM$ the other bits.

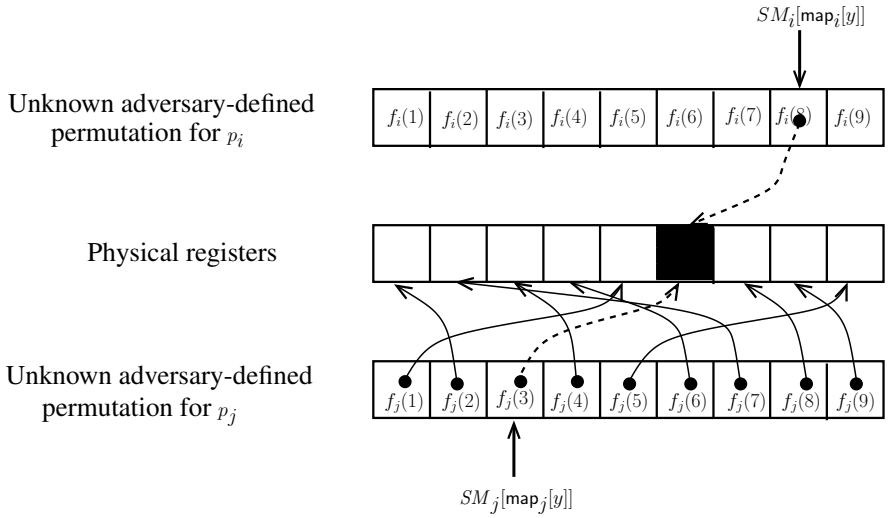

Fig. 1. An example of de-anonymization, $n = 4$ and $m = 2n + 1$

The meaning and use of $SM[x].RM$ are exactly the same as $SM[x]$ in Algorithm 1 and Algorithm 2. For each x, $SM[x].BIT$ is initialized to 0, while $SM[x].RM$ is initialized to $\langle \mathtt{start}, \perp \rangle$. We assume that the previous algorithms are appropriately updated so that they do not modify the bits $SM[x].BIT$.
Not to overload the presentation, the following notation shortcuts are used in Algorithm 3.

- The read of $SM_i[x]$ at lines 3 and 4 concerns the field $SM_i[x].RM$.
- The write of $SM_i[x]$ at lines 2 and 4 writes 0 in its leftmost bit (which actually is not modified).
- The statement "$BIT_i[x] \leftarrow 1$" at line 6, means that only the leftmost bit of $SM_i[x]$ is modified. As this statement is issued by the leader process only, this process can first read $SM_i[x]$, prefix it by 1, and rewrite this new value so that only the leftmost bit $SM_i[x]$ is modified.
- The statement "$BIT_i.\mathsf{scan}()$" stands for "$SM_i.\mathsf{scan}()$" from which only the leftmost bits are extracted.

After they return from de-anonymize(), the processes execute the same synchronization pattern as lines 14-17 of Algorithm 1 where the tag `start` is replaced by the tag `desa`. As the reader can see, at this time the tag `done` is no longer present in a register, so it can be re-used. Moreover, as the type "process identity" and the type "integer" are different, any integer x is considered as a synonym of $\perp$ when looking at a pair $\langle \mathtt{desa}, x \rangle$ (which now is a synonym of $\langle \mathtt{start}, x \rangle$).

It follows that we have then the property P1': there is a time at which exactly one register contains $\langle \mathtt{start}, z \rangle$ where z is an integer and, for each $j \in \{1, \cdots, n\}$, α registers contain $\langle \mathtt{start}, id_j \rangle$. Here, the important point is that the process previously

operation SM_i.scan() **returns** $([SM_i[1], \cdots, SM_i[m]])$.

operation efficient_de-anonymize(id_i) **is**
(01) de-anonymize(id_i);
% As all reg. are tagged `desa` when the first process returns from de-anonymize()
% the tags `start` and `done` disappeared from the system and can be re-used
(02) execute lines 1-10 of Algorithm 1 where `start` is replaced by `desa`;
% in the following, ℓ_i has the value obtained in de-anonymize(id_i)
(03) $SM_i[\ell_i] \leftarrow \langle \texttt{done}, id_i \rangle$;
(04) wait until$\big((SM_i[\ell_i] \neq \langle \texttt{done}, id_i \rangle)$
$\vee$ ($SM_i[1..m]$ has exactly $\alpha + 1$ entries not tagged `done`)$\big)$;
(05) **for each** x such that $SM_i[x] = \langle \texttt{desa}, id_i \rangle$ **do** $SM_i[x] \leftarrow \langle \texttt{done}, id_i \rangle$ **end for**;
% Property P1': There is a time at which exactly one register contains $\langle \texttt{start}, z \rangle$
% where z is an integer and for each $j \in \{1, \cdots, n\}$, α registers contain $\langle \texttt{start}, id_j \rangle$
(06) **if** $(leader_i = id_i)$
% Here the leader knows that every process p_j knows its mapping function $\mathsf{map}_j()$
(07) **then for each** $x \in \{1, \cdots, m\}$ **do** $BIT_i[x] \leftarrow 1$ **end for**
(08) **else repeat** $bit_i \leftarrow BIT_i$.scan() **until** $(\forall\, x :\ bit_i[x] = 1)$ **end repeat**
(09) **end if**.

Algorithm 3: Reduction to a single bit of control information per register (code for p_i)

elected as a leader knows that any process p_j knows its mapping function $\mathsf{map}_j()$. So, it can inform of it the other processes. This is done at lines 6-9 of Algorithm 3. As soon as a process p_j sees the leftmost bit of all the registers equal to 1, it knows that each process knows its mapping function, and p_j can consequently start writing application-related values in the other bits of the registers.

The lines 2-9 of Algorithm 3 and the code of Algorithm 1 are nearly the same. More precisely, they differ in the fact that Algorithm 1 *elects* a leader at lines 13-14, while Algorithm 3 uses at line 3 the leader that has been *previously been elected*. It follows that the proof of Algorithm 3 is very close to the proof of Algorithm 1, and is left to the reader.

5 Memory Anonymous Leader Election when $m = \alpha\, n + (n-1)$

Leader election. Algorithm 1, which solves the election problem for a system of $m = \alpha\ n + 1$ anonymous registers, is 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 remaining not yet written anonymous register is used to elect the leader (which will be the last process that writes its identity in this register well-identified by each process).

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

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

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) $written_i \leftarrow (written_i \setminus overwritten_i) \cup towrite_i$;
(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) $towrite_i \leftarrow \{last_i + 1, ..., \mathsf{min}(last_i + nb_i, m)\}$;
(11) $last_i \leftarrow \mathsf{min}(last_i + nb_i, m)$
(12) **end if**
(13) **end repeat**;
% Property P1": There is a time at which α reg. contain the same pair $\langle start, id_\ell\rangle$,
% and for each $j \in \{1, \cdots, n\} \setminus \{\ell\}$, $\alpha+1$ registers contain $\langle \texttt{start}, id_j\rangle$
(14) $\texttt{leader}_i \leftarrow id$ **where** id
(15) **is such that** α registers exactly contain the same pair $\langle start, id\rangle$.

Algorithm 4: n-process election for $m = \alpha\, n + (n-1)$ anonymous registers

Algorithm. The operational view of this idea is captured by Algorithm 4, obtained from a simple adaptation of Algorithm 1 to the fact that the leader is selected from a memory occupation criterion (instead of a competition on a single read/write register, where the last writer is the winner). The main difference lies in the management of the local variables $towrite_i$, $overwritten_i$, $written_i$, $last_i$, and nb_i. Property P1" captures the result of the algorithm, namely, there is a time at which α registers contain the same pair $\langle start, id_\ell\rangle$, and for each $j \in \{1, \cdots, n\} \setminus \{\ell\}$, $\alpha+1$ registers contain $\langle \texttt{start}, id_j\rangle$. Its proof is a simple adaptation of the proof of Algorithm 1.

6 Election and De-anonymization for $m = \alpha\, n + \beta, \beta \in M(n)$

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

Mutual exclusion in an anonymous system. Mutual exclusion in memory anonymous systems was introduced in [15], which presents a symmetric deadlock-free mutex algorithm for *two* processes only, and a theorem stating that there 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. The idea is to rely on the underlying mutex algorithm to elect a leader. But, to this end, the processes have first to isolate a set of β anonymous registers in order to be thereafter able to use a symmetric deadlock-free mutex algorithm accessing this subset of registers.

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

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$\big((\exists\, x \in written_i :\ SM_i[x] \neq \langle \texttt{start}, id_i \rangle)$
$\vee\ (|\{\ell \text{ such that } SM_i[\ell] = \langle \texttt{start}, \bot\rangle\}| = \beta)\big)$;
(06) **if** $\big(|\{\ell \text{ such that } SM_i[\ell] = \langle \texttt{start}, \bot\rangle\}| = \beta\big)$
(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) $towrite_i \leftarrow \{last_i + 1, ..., last_i + nb_i\}$; $last_i \leftarrow last_i + nb_i$
(11) **end if**
(12) **end repeat**;
% Property P1"': There is a time at which β registers contain the pair $\langle start, \bot\rangle$,
% and for each $j \in \{1, \cdots, n\}$, α registers contain $\langle \texttt{start}, id_j\rangle$
(13) **let** $SM\beta_i[1..\beta]$ be the sub-array of the β registers
that do not contain $\langle \texttt{start}, id\rangle$, for any process identity id;
(14) Now, using the previous sub-array (locally knows as $SM\beta_i[1..\beta]$ by p_i) the processes 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 all the registers of $SM\beta_\ell[1..\beta]$, which allows the other processes to know which is the leader.

Algorithm 5: Election in a system of $m = \alpha\ n + \beta$, $\beta \in M(n)$ anonymous reg.

Algorithm 5 realizes this at lines 1-12, which are a simple adaptation of the same line numbers in Algorithm 1 and Algorithm 4. When the processes exit the repeat loop (line 12), we have property P1"', namely, there is a time at which β registers contain the pair $\langle start, \bot\rangle$ and, for each $j \in \{1, \cdots, n\}$, α registers contain $\langle \texttt{start}, id_j\rangle$. Hence, the set of β registers define a common anonymous memory on top of which the n processes 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. It is shown in [7] 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. Finally, the last process to enter is the elected process.

We point out that a memory de-anonymization algorithm is described in [7]. However, as it is based on an underlying mutual exclusion algorithm, it is a specific algorithm

that works only for $m \in M(n)$, which is not the general case addressed here, namely $m = \alpha\, n + \beta$.

Memory de-anonymization in a system of $m = \alpha\, n + \beta$ anonymous registers. The previous algorithm can be modified in order to solve de-anomymization. When the last process is inside the critical section, it can impose its mapping function to all the processes by executing lines 4-5 of Algorithm 2, while all the other processes execute lines 5-6 of this algorithm.

7 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 memory anonymous sizes for which symmetric deadlock-free mutual exclusion can be solved in n-process systems. De-Anonymization algorithms have also been presented, each based on an underlying election algorithm.

As stated in [15], the memory-anonymous 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. Additional results regarding the computational power of anonymous and non-anonymous objects can be found in [16]. On a more practical side, it appears that the concept of an anonymous memory allows us to model epigenetic cell modifications [12].

On the open problems side, it seems that finding a characterization of all the values of m (the size of the read/write anonymous memory) for which leader election (and de-anonymization) can be solved in an n-process system is particularly important as soon as we want to understand the power and the limits of n-process memory anonymous systems. Finally, since we assume a model where participation is required, in the case where the mutex algorithm from [2] (which also works when participation is not required) is used, it might be possible to replace the algorithm from [2] with a simpler algorithm. In such a case we might not need to assume that $\beta \in M(n)$, but something weaker.

Acknowledgments

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

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, 10 pages (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. Bonnet F. and Raynal M., Anonymous asynchronous systems: the case of failure detectors. *Distributed Computing*, 26(3):141-158 (2013)
5. 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)
6. 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)
7. 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'18)*, To appear, Springer LNCS, 15 pages (2019)
8. Guerraoui R. and Ruppert E., Anonymous and fault-tolerant shared-memory computations. *Distributed Computing*, 20:165-177 (2007)
9. 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)
10. Raynal M., *Concurrent programming: algorithms, principles and foundations*. Springer, 515 pages, ISBN 978-3-642-32026-2 (2013)
11. 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)
12. Rashid S., Taubenfeld G., and Bar-Joseph Z., Genome wide epigenetic modifications as a shared memory consensus. *6th Workshop on Biological Distributed Algorithms (BDA'18)*, London (2018)
13. Styer E., and Peterson G. L. Tight bounds for shared memory symmetric mutual exclusion problems. In *Proc. 8th ACM Symposium on Principles of Distributed Computing*, ACM Press, pp. 177-191 (1989)
14. Taubenfeld G., *Synchronization algorithms and concurrent programming*. Pearson Education/Prentice Hall, 423 pages, ISBN 0-131-97259-6 (2006)
15. Taubenfeld G., Coordination without prior agreement. *Proc. 36th ACM Symposium on Principles of Distributed Computing (PODC'17)*, ACM Press, pp. 325-334 (2017)
16. 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, 15 pages (2019)
17. 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)