Mutual Exclusion in Fully Anonymous Shared Memory Systems

Michel Raynal[⋆,◇], Gadi Taubenfeld[◦]

[⋆]Univ Rennes IRISA, Inria, Cnrs, France
[◇]Department of Computing, Polytechnic University, Hong Kong
[◦]The Interdisciplinary Center, Herzliya, Israel

Abstract

Process anonymity has been studied for a long time. Memory anonymity is more recent. In an anonymous memory system, there is no a priori agreement among the processes on the names of the shared registers. 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 register X, while the same name C may correspond to different register names for the processes p and q, and this remains unknown to the processes. This article introduces the *full anonymous* model, namely a model in which both the processes and the registers are anonymous. A fundamental question is then "is this model meaningful?", which can be translated as "can non-trivial fundamental problems be solved in such a very weak computing model?"

This article answers this question positively. More precisely, it presents a deadlock-free mutual exclusion algorithm in such a fully anonymous model where the anonymous registers are read/modify/write registers. This algorithm assumes that m (the number of shared registers) and n (the number of processes) are such that m is relatively prime with all the integers $\ell \in \{1, ..., n\}$. Combined with a previous result (PODC 2019) on mutual exclusion in memory anonymous (but not process anonymous) systems, it follows that this condition is both necessary and sufficient for the existence of such an algorithm in fully anonymous systems. As far as we know, this is the first time full anonymity is considered, and where a non-trivial concurrency-related problem is solved in such a very strong anonymity context.

Keywords: Anonymous memory, Anonymous processes, Mutual exclusion.

1 Introduction: Computing Model

1.1 On the process side

Process anonymity 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, 24]) and shared memory systems (e.g., [3, 6, 10]). Process anonymity means that processes have no identity, have the same code and the same initialization of their local variables (otherwise they could be distinguished). Hence, in a process anonymous system, it is impossible to distinguish a process from another process.

Process model The system is composed of a finite set of $n \geq 2$ asynchronous, anonymous sequential processes denoted p_1, .., p_n. Each process p_i knows the number of processes n and the total number of registers m. The subscript i in p_i is only a notational convenience, which is not known by the processes.

Sequential means that a process executes one step at a time. *Asynchronous* means that each process proceeds in its own speed, which may vary with time and always remains unknown to the other processes.

1.2 On the memory side

Memory anonymity The notion of *memory anonymity* has been recently introduced in [22]. Let us consider a shared memory R made up of m atomic registers. Such a memory can be seen as an array with m entries, namely $R[1..m]$. In a non-anonymous memory system, for each index x, the name $R[x]$ denotes the same register whatever the process that accesses the address $R[x]$. Hence in a non-anonymous memory, there is an a priori agreement on the names of the shared registers. This facilitates the implementation of the coordination rules the processes have to follow to progress without violating the safety properties associated with the application they solve [11, 19, 21].

The situation is different in an anonymous memory, where there is no a priori agreement on the name of each register. Moreover, all the registers of an anonymous memory are assumed to be initialized to the same value (otherwise, their initial values could provide information allowing processes to distinguish them). The interested reader will find an introductory survey on process and memory anonymity in [20].

Anonymous shared memory The shared memory is made up of $m \geq 1$ atomic anonymous registers denoted $R[1...m]$. Hence, *all* the registers are anonymous. As already indicated, due to its anonymity, $R[x]$ does not necessarily indicate the same object for different processes. More precisely, a memory-anonymous system is such that:

- For each process p_i an adversary defined a permutation $f_i()$ over the set $\{1, 2, \cdots, m\}$, such that when p_i uses the address $R[x]$, it actually accesses $R[f_i(x)]$,
- No process knows the permutations, and
- All the registers are initialized to the same default value denoted $\perp$.

identifiers for an external observer	local identifiers for process p_i	local identifiers for process p_j
$R[1]$	$R_i[2]$	$R_j[3]$
$R[2]$	$R_i[3]$	$R_j[1]$
$R[3]$	$R_i[1]$	$R_j[2]$
permutation	$f_i() : [2, 3, 1]$	$f_j() : [3, 1, 2]$

Table 1: Illustration of an anonymous memory model

An example of anonymous memory is presented in Table 1. To make apparent the fact that $R[x]$ can have a different meaning for different processes, we write $R_i[x]$ when p_i invokes $R[x]$.

Anonymous register model We consider here the read/modify/write (RMW) model. In this model, each register can be read, written or accessed by a conditional write operation that atomically reads the register and (according to the value read) possibly modifies it. More precisely, this operation, denoted `compare&swap`(Z, old, new), has three input parameters, a shared register Z and two values old and new, and returns a Boolean value. It has the following effect: if $Z = old$ the value new is assigned to Z

and the value `true` is returned (the `compare&swap()` operation is then successful). If $Z \neq old$, Z is not modified, and the value `false` is returned.

Atomicity [13] means that the operations on the shared registers appear as if they have been executed sequentially, each operation appearing between its start event and its end event, and for any $x \in \{1, ...m\}$, a6 read operation of a register $R[x]$ returns the value v, where v is the last value written in $R[x]$ by a write or a successful `compare&swap`$(R[x], -, -)$ operation (we also say that the execution is *linearizable* [12]).

1.3 Motivation

This article addresses the mutual exclusion problem in fully anonymous systems and 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 processes and an anonymous memory, and providing associated necessary and sufficient conditions, helps us determine the weakest system assumptions under which fundamental problems, such as mutual exclusion can be solved.

The second one is application-oriented. It has been shown in [18] 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. Hence, fully anonymous shared memories could be useful in biologically inspired distributed systems [14, 15]. Thus, mastering a fundamental problem such as mutual exclusion in such an adversarial context could reveal to be important from an application point of view.

1.4 Mutual exclusion

Mutual exclusion is the oldest and one of the most important synchronization problems. Formalized by E.W. Dijkstra in the mid-sixties [8], it consists of building what is called a lock (or mutex) object, defined by two operations, denoted acquire() and release().

The invocation of these operations by a process p_i follows the following pattern: "acquire(); *critical section*; release()", where "critical section" is any sequence of code. It is assumed that, once in the critical section, a process eventually invokes release(). A mutex object must satisfy the following two properties.

- Mutual exclusion: No two processes are simultaneously in their critical section.
- Deadlock-freedom progress condition: If there is a process p_i that has a pending operation acquire() (i.e., it invoked acquire() and its invocation is not terminated) and there is no process in the critical section, there is a process p_j (maybe $p_j \neq p_i$) that eventually enters the critical section.

Two memory-anonymous (but not process-anonymous) symmetric deadlock-free mutual exclusion algorithms are presented in [2]. One is for the RW register model (in which only atomic read and atomic write operations on registers are supported), the other one for the RMW register model. These two algorithms are symmetric in the sense that the processes have identities that can only be compared for equality. We notice that algorithms for anonymous processes are, by definition, symmetric.

Mutual exclusion cannot be solved in the presence of process crash failures: if a process crashes just after it obtained the critical section, it will never release it, and consequently, the upper layer application can block forever. The computing model must be enriched with additional computability power (for example with failure detectors, see e.g., [4, 7]) to be able to solve mutual exclusion in the presence of failures.

Let us point out that, in a model which supports only atomic read and atomic write operations on registers, there is no mutual exclusion algorithm when the processes are anonymous[1], be the registers anonymous or not. To see that, simply consider an execution in which the anonymous processes run in a lock-step manner. *Lock-step* means that the processes execute as follows. First, each process -one after the other- executes its first operation. As the processes are anonymous, their first operation is the very same read or write (of the same value) on the same register. Hence, after this first lock-step occurred, all the processes are in the same local state. Then, in the same order as before, the processes execute their second operation. As before, at the end of this second lock-step, the processes are in the same local state. Etc. In such a run it is not possible to break symmetry as the local states of the processes are exactly the same after each lock-step.

2 A Fully Anonymous RMW Mutex Algorithm

The principle on which relies the RMW fully anonymous mutex algorithm presented in this section is different from the one on which relies the RMW memory-anonymous (but not process anonymous) mutex algorithm presented in [2].

The anonymous memory As already indicated, each RMW register of the anonymous memory $R[1..m]$, is initialized to the value $\perp$. Moreover, it is assumed that $\perp$ is smaller than any non-negative integer.

Let $M(n)$ denote the set of all the integers m such that, for all $\ell \in \{1, ..., n\}$, ℓ and m are relatively prime (i.e., $\mathtt{gcd}(\ell, m) = 1$). It is assumed that the size m of the shared memory is such that $m \in M(n)$. This assumption will be justified in Section 3 and Section 4.

Local variables at each process Each process p_i manages the following local variables.

- max_i is used to store the maximal value contained in a register (as seen by p_i).
- $counter_i$ is used to store the number of registers *owned* by p_i. A process *owns* a register when it is the last process that wrote a non-$\perp$ value into this register.
- $myview_i[1..n]$ is an array of Boolean values, each initialized to `false`. When $myview_i[j]$ is equal to `true`, p_i owns the register $R_i[j]$.
- $round_i$ (initialized to 0) is the round number (rung number in the ladder metaphor, see below) currently attained by p_i in its competition to access the critical section. When $round_i = n$, p_i is the winner and can enter the critical section.

Principle of the algorithm: concurrent climbing of a narrowing ladder At some abstract level, the principle that underlies the behavior of the algorithm is simple[2]. Assume there is a ladder with $(n + 1)$ rungs, numbered form 0 to n. Initially, all the processes are at rung number 0 (hence their local variables $round_i$ are equal to 0). For each process p_i, $round_i$ is equal to the rung number it attained. The aim of the algorithm is to allow processes to progress from a rung r to the next rung $(r + 1)$ of the ladder, while ensuring that, for any $r \geq 1$, at most $(n - r + 1)$ processes currently are at rung r. From the local point of view of a process, this means that process p_i is allowed to progress to the rung $r = round_i + 1$ only when some specific condition is satisfied. This condition involves the notion of *ownership* of an anonymous register (see above), and the asymmetry seed provided by the model, namely $m \in M(n)$.

[1] Let us remind that, as they are anonymous, the processes have the same code and the same initialization of their local variables.

[2] This principle is not new. As an example it is found in Peterson's n-process RW mutex algorithm, where processes raise and lower individual flags –visible by all processes– and write their identity in a size n non-anonymous memory [16].

Algorithm The algorithm is described in Fig. 1. A process enters a "repeat" loop, that it will exit when it will have attained the last rung of the ladder, i.e., when $round_i = n$. When $round_i = r > 0$, which means p_i is at round r, it strives to own more registers, by writing the rung number r in the registers it owned previously and in new registers. Its behavior in the loop body is composed of three parts.

- Part 1: lines 2-6. A process p_i first scans (sequentially and asynchronously) all the registers to know the highest value they contain. This value is stored in max_i (lines 3-4). Then, if no register was equal to $\perp$ when p_i read it (i.e., from p_i's point of view, they all were owned by some processes) we have then $round_i < max_i$ at line 5, and p_i remains at round 0, which means it loops at lines 3-6 until it finds all the registers equal to $\perp$. In short, as p_i sees that other processes climbed already at higher rungs, it stays looping at the rung numbered 0.
- Part 2: lines 7-17. This part subdivides in two sub-parts, according to the round number of p_i. In both cases, p_i tries to own as many registers as possible.
  - $round_i = 1$. In this case, p_i owns no registers. So, it scans the anonymous memory and, for each register $R_i[j]$, it invokes `compare&swap`$(R_i[j], \perp, 1)$ to try to own it. If it succeeds, it updates $myview_i$ and $counter_i$ (line 8-10).
  - $round_i \geq 2$. In this case, p_i became the owner of some registers during previous rounds. It then confirms its ownership of these registers with respect to its progress to the current round r (line 12-13). Then it strives to own more registers. But, to ensure deadlock-freedom, it considers only the registers that contain a round number smaller than its current round r. The array $myview_i$ and the local variable $counter_i$ are also updated according to the newly owned registers (line 14-17).
- Part 3: lines 18-24. The aim of this part is to ensure deadlock-freedom. As the proof will show, if p_i attains rung $r > 0$ (i.e., $round_i = r$), there are at most $(n - r + 1)$ processes competing with p_i (line 19), and these processes attained a rung $\geq r$. In this case, at least one of them (but not all) must withdraw from the competition so that at most $(n - r)$ processes compete for the rung r.

  The corresponding "withdrawal" predicate is $counter_i < m/(n - r + 1)$ (line 20), which involves the asymmetry-related pair (n, m) and $round_i = r$, which measures the current progress of p_i. If the withdrawal predicate is false and p_i attained $round_i = n$, it enters the critical section (predicate of line 25). If the predicate is false and $round_i < n$, p_i re-enters the loop, to try to own more registers and progress to the next rung of the ladder.

  If the withdrawal predicate is true, p_i releases all the registers it owns and updates $myview_i$ accordingly (lines 21-22). Then, it waits until it sees all the registers equal to their initial value (lines 23). After that, p_i resets its local variables to their initial values (lines 24), and re-enters the loop body.

3 Proof of the Algorithm

Reminder: $M(n) = \{m$ such that $\forall \ell : 1 \leq \ell \leq n : \mathsf{gcd}(\ell, m) = 1\}$. Moreover, let us say that "process p_i executes round r" when its local variable $round_i = r$.

Lemma 3.1 *Let $m \in M(n)$ and $r \in \{2, ..., n\}$. The values $m/(n - r + 1)$ are not integers.*

Proof The set of the values $(n - r + 1)$ for $r \in \{1, ..., n - 1\}$ is $X = \{n, n - 1, ..., 2\}$. The fact that, for any $x \in X$, m/x is not an integer is a direct consequence of the definition of m, namely, $m \in M(n)$.
$\square_{Lemma\ 3.1}$

ALGORITHM 1: CODE OF AN ANONYMOUS PROCESS p_i

Constants:
n, m: positive integers, // # of processes and # of shared registers
model constraint // $\forall\ \ell:\ 1 \leq \ell \leq n,\ m$ and ℓ are relatively prime
Anonymous RMW shared registers:
$R[1..m]$: array of m anonymous RMW registers, initially all $\perp$ // $\perp < 0$
Local variables:
$myview_i[1..m]$: array of m Boolean bits, initially all `false` // indicates ownership
$counter_i, round_i, max_i$: integer

operation acquire() **is**
1 $counter_i \leftarrow 0; round_i \leftarrow 0$ // begin entry code
2 **repeat**
3 $max_i \leftarrow 0$ // check if another process is in a higher round
4 $max_i \leftarrow \mathsf{max}(max_i, R_i[1], \ldots, R_i[m])$ // find maximum in $R_i[1..m]$
5 **if** $round_i < max_i$ **then** $round_i \leftarrow 0$ // withdraw from the competition
6 **else** $round_i \leftarrow round_i + 1$ **fi** // continue to the next round

7 **if** $round_i = 1$ **then** // first round
8 **for each** $j \in \{1, ..., m\}$ **do** // try to own as many shared
9 $myview_i[j] \leftarrow$ `compare&swap`$(R_i[j], \perp, 1)$ // registers as possible
10 **if** $myview_i[j]$ **then** $counter_i \leftarrow counter_i + 1$ **fi od fi** // own one more

11 **if** $round_i \geq 2$ **then** // try to own additional released registers
12 **for each** $j \in \{1, ..., m\}$ **do**
13 **if** $myview_i[j]$ **then** $R_i[j] \leftarrow round_i$ **fi od** // update all owned registers
14 **for each** $j \in \{1, ..., m\}$ **do**
15 **while** $R_i[j] < round_i$ **do** // $R[j] < round_i$ implies $myview[j] =$ `false`
16 $myview_i[j] \leftarrow$ `compare&swap`$(R_i[j], \perp, round_i)$ // try to own $R_i[j]$
17 **if** $myview_i[j]$ **then** $counter_i \leftarrow counter_i + 1$ **fi od fi** // own one more

18 **if** $round_i \geq 1$ **then** // not eliminated
19 $competitors \leftarrow n - round_i + 1$ // max # of competing processes
20 **if** $counter_i < m/competitors$ **then** // withdraw from the competition
21 **for each** $j \in \{1, ..., m\}$ **do** // since not own enough registers
22 **if** $myview_i[j]$ **then** $R_i[j] \leftarrow \perp$; $myview_i[j] \leftarrow$ `false` **fi od** // release
23 wait$(\forall\ j \in \{1, ..., m\}:\ R_i[j] = \perp)$; //wait until all are $= \perp$
24 $counter_i \leftarrow 0; round_i \leftarrow 0$ **fi** // start over
25 **until** $round_i = n$ // until the winner owns all m registers
26 return(`done`).

operation release() **is**
27 **for each** $j \in \{1, ..., m\}$**do** $R_i[j] \leftarrow \perp$; $myview_i[j] \leftarrow$ `false` **od** // release all
28 return(`done`).

Figure 1: Deadlock-free mutual exclusion for n anonymous processes and $m \in M(n)$ anonymous RMW registers

Lemma 3.2 *Let us consider the largest round r executed by processes. At most $(n - r + 1)$ processes are executing a round r.*

Proof Let us consider a process that executes line 6, where it sets its local variable $round_i$ to 1. As there are n processes, trivially at most n processes are simultaneously executing round $r = 1$. Let us assume (induction hypothesis) that round r is the largest round currently executed by processes, and at most $(n - r + 1)$ processes execute it. We show that at most $(n - r)$ processes will execute round $r + 1$.

Let P_r be the set of processes that execute round r. Let us consider the worst case, namely, $|P_r| = n - r + 1$. We have to show that at least one process of P_r will not execute round $(r + 1)$. This amounts to showing that at least one process p_i of P_r never exits the wait statement of line 23, or executes line 24 where it resets its variable $round_i$ to 0. Whatever the case, this amounts to showing that there is at least one process p_i of P_r for which the predicate $counter_i < m/(n - r + 1)$ is satisfied at line 20.

When a process of P_r exits the set of statements of lines 8-10 when $r = 1$, or line 12-17 when $r > 1$, the value of each anonymous register is $\geq r$. Let us observe that, when different from 0, the local variable $counter_i$ of a process p_i counts the number of anonymous registers that this process set equal to $round_i$, where $round_i = r$, i.e., $counter_i = |\{x$ such that $myview_i[x] = \texttt{true}\}|$ (line 8-10 when $round_i = 1$, and lines 12-17 when $round_i > 1$). Notice also that, in the last case, $counter_i$ increases from round to round and thanks to the atomicity of the operation $\texttt{compare\&swap}(R[j], \perp, round_i)$ at line 9 or 16 that, with respect to the registration in the local variables $myview_i[1..n]$, no anonymous register can be counted several times by the same process or counted by several processes.

Assume (by contradiction) that the predicate of line 20 is false at each process of P_r, and let $counter(x)$, for $1 \leq x \leq |P_r|$, be the value of their counter variables. Then $counter(1) + \cdots + counter(|P_r|) = m$, and (due to the assumption that the predicate of line 20 is false) each counter is greater or equal to $m/(n-r+1)$. Hence, $\forall x :\ counter(x) \geq m/(n - r + 1)$. As, due to Lemma 3.1, $m/(n - r + 1)$ is not an integer, it follows that $\forall x :\ counter(x) \geq \lceil m/(n-r+1) \rceil$. And consequently, $counter(1) + \cdots + counter(|P_r|) \geq (n - r + 1)\lceil m/(n - r + 1) \rceil$. But $(n - r + 1)\lceil m/(n - r + 1) \rceil > m$, a contradiction.

Hence, at least one local variable $counter$ is such that $counter < (m/(n - r + 1))$. It follows that at least one process of P executes line 27, which concludes the proof of the lemma. $\square_{Lemma\ 3.2}$

Lemma 3.3 *No two processes are simultaneously in the critical section.*

Proof The lemma follows directly from the previous lemma and the fact that a process enters the critical section only when its local variable $round = n$ (line 25). $\square_{Lemma\ 3.3}$

Lemma 3.4 *Let r, $1 \leq r < n$, be the highest round attained by processes. At least one process attains the round $(r + 1)$.*

Proof Let r, $1 \leq r < n$, be the highest round attained by processes, and $P(r)$ the corresponding set of processes. As in the proof of Lemma 3.2, let P_r be the set of processes that execute round r. As previously, we have $counter(1) + \cdots + counter(|P_r|) = m$. If the predicate of line 20 is satisfied at each process of P_r we have $\forall x :\ counter(x) < m/(n - r + 1)$. As due to Lemma 3.1 $m/(n - r + 1)$ is not an integer, it follows that $\forall x :\ counter(x) \leq \lfloor m/(n - r + 1) \rfloor$. Consequently, $counter(1) + \cdots + counter(|P_r|) \leq (n - r + 1)\lfloor m/(n - r + 1) \rfloor$. But $(n - r + 1)\lfloor m/(n - r + 1) \rfloor < m$, a contradiction. $\square_{Lemma\ 3.4}$

Lemma 3.5 *If at some time no process is inside the critical section and one or more processes want to enter the critical section, at least one process will enter it.*

Proof The lemma follows directly from the previous lemma, applied from round 1 to round n. $\square_{Lemma\ 3.5}$

The following theorem is a direct consequence of Lemma 3.3 and Lemma 3.5.

Theorem 3.6 *Assuming $m \in M(n)$, Algorithm 1 solves deadlock-free mutual exclusion in a fully anonymous system.*

4 On the Computability Side

It follows from Theorem 3.6 that $m \in M(n)$ is a sufficient condition for deadlock-free mutual exclusion in a fully anonymous system. On another side, the lower bound result in [2] states that $m \in M(n)$ is a necessary and sufficient condition for symmetric deadlock-free mutual exclusion for n non-anonymous processes and m anonymous RMW registers. "Symmetric" has been defined in Section 1.4 where it was pointed out that algorithms for anonymous processes are inherently symmetric.

As the non-anonymous processes and anonymous RMW registers model is at least as strong as the fully anonymous RMW model, it follows from the previous observations that $m \in M(n)$ is a necessary and sufficient condition for deadlock-free mutex in fully anonymous systems.

Hence, as far as symmetric deadlock-free mutual exclusion using RMW registers is concerned, the necessary and sufficient condition $m \in M(n)$ shows that there is no computability gap between full anonymity (as addressed here) and register-restricted anonymity [2]. Actually, this condition tightly captures the initial "asymmetry-breaking seed" that allows n (anonymous or non-anonymous) processes to solve deadlock-free mutex on top of an anonymous memory made up of m RMW registers.

5 Related Work

Early work While the work presented previously considers that both processes and registers are anonymous, the case where only the registers are anonymous was implicitly introduced by by Michael O. Rabin in solving a problem he called *Choice Coordination Problem with k alternatives* [17]. In this problem, n processes must choose the same alternative 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.

Foundational results The notion of anonymous registers was formally defined in [22], where 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 given. 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.

The open problem from [22], regarding the existence of a memory-anonymous two-process starvation-free mutual exclusion algorithm is still open. Leader election on top of anonymous read/write registers de-anonymization is addressed in [9].

Results regarding the computational power of anonymous and non-anonymous registers can be found in [23], where it is proved that anonymous bits are strictly weaker than anonymous (and hence also non-anonymous) multi-valued registers.

Algorithms in nature Anonymous shared memory systems are useful in modeling biologically inspired distributed computing methods, especially those that are based on ideas from molecular biology [14, 15, 18]. For example, as was already pointed out in the Introduction, the concept of an anonymous memory allows epigenetic cell modification to be modeled from a distributed computing point of view [18].

6 Conclusion

This article has introduced the notion of *fully anonymous* shared memory systems, namely, systems where not only the processes are anonymous but the shared memory also is anonymous in the sense that there is no global agreement on the names of the shared registers (any register can have different names for distinct processes).

From a technical point of view, it has then presented a non-trivial mutual exclusion algorithm designed for this computation model, where the anonymous registers are read/modify/write registers. It has also shown that mutex can be solved despite full anonymity if and only if the pair (n, m) of system parameters (number of anonymous processes, size of the anonymous shared memory) is such that m is relatively prime with all integers in the set $\{1, ..., n\}$. This condition constitutes the weakest asymmetry seed from which mutex can be solved in the full anonymity context.

Acknowledgments

The authors want to thank the referees for their careful reading and their constructive comments that helped improve the presentation. M. Raynal 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, 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] Bhatt V. and Jayanti P., On the existence of weakest failure detectors for mutual exclusion and k-exclusion. *23rd Int'l Symposium on Distributed Computing (DISC'09)*, Springer LNCS 5805, pp. 325-339 (2009)

[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] Delporte C., Fauconnier H., and Raynal M., On the weakest failure detector for read/write-based mutual exclusion. *Proc. 33nd Int'l Conference on Advanced Information Networking and Applications (AINA'19)*, Springer AICS 926, pp. 272-285 (2019)

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

[9] Godard E., Imbs D., Raynal M., and Taubenfeld G., Anonymous read/write memory: leader election and desanonymization. *Proc. 26th Intl Colloquium on Structural Information and Communication Complexity (SIROCCO19)*, Springer LNCS 11639, pp. 246-261 (2019)

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

[11] Herlihy M. and Shavit N., *The art of multiprocessor programming*. Morgan Kaufmann, 508 pages, ISBN 978-0-12-370591-4 (2008)

[12] Herlihy M.P. and Wing J.M, Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492 (1990)

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

[14] 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)

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

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

[17] Rabin M. O., The choice coordination problem. *Acta Informatica*, 17:121134 (1982)

[18] Rashid S., Taubenfeld G., and Bar-Joseph Z., Genome-wide epigenetic modifications as a shared memory consensus. *6th Workshop on Biological Distributed Algorithms (BDA18)*, London (2018)

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

[20] 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)

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

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

[23] 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 (SIROCCO19)*, Springer LNCS, pp. 293-308 (2019)

[24] 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)