The Distributed Computing Column

Seth Gilbert
National University of Singapore
seth.gilbert@comp.nus.edu.sg

In this issue of the distributed computing column, Michel Raynal and Gadi Taubenfeld revisit a classical question: what can we accomplish in shared memory systems that are anonymous and symmetric? The article is a nice introductory presentation of symmetry and anonymity. It differentiates symmetry and anonymity in the context of processors, and looks at both process and memory anonymity. The article uses two fundamental problems to illustrate: mutual exclusion and consensus. In each case, it illustrates what can be accomplished, giving a good illustration of both what is feasible and the limitations in symmetric and anonymous systems. This new distributed computing column is a great way to start the new year, looking back at a classic problem!

Symmetry and Anonymity in Shared Memory Concurrent Systems

Michel Raynal[⋆], Gadi Taubenfeld[◇]

[⋆]Univ Rennes IRISA, Inria, CNRS, 35042 Rennes, France
[◇]The Interdisciplinary Center, Herzliya, Israel

1 Introduction

More than forty years ago, Dana Angluin asked the following question at one of the first symposia on the theory of computing (STOC 1980) [2]:

> "How much does each processor in a network of processors need to know about its own identity, the identities of other processors, and the underlying connection network in order for the network to be able to carry out useful functions?"

If addressing computability issues was mainly perceived as a theoretical question in 1980, due to the fantastic boom in the development of concurrent and distributed systems (whether the communication is through shared memory or message-passing), today this question is becoming more and more important. In such a context, this technical article visits three important anonymity-related notions. The first two concern restrictions on the identities of the processes (namely, symmetry and anonymity). The third one is relatively new: it concerns the notion of an anonymous shared memory [32].

2 Process Symmetry

The notion of *symmetry* is pervasive in many domains, from philosophy and arts (mainly architecture and painting) to scientific areas such as physics, chemistry, and mathematics [12]. In informatics, a form of process symmetry (related to fairness) was introduced by Edsger W. Dijkstra in his famous one-page article that presented the mutual exclusion problem and a solution to it [10]. More precisely, when he defined the problem, Dijkstra wrote, "The solution must be symmetrical

between the n computers; as a result we are not allowed to introduce a static priority." More than twenty years later (1989), the notion of *process symmetry* was explicitly defined and investigated by Eugene Styer and Gary L. Peterson in an article presented at the ACM conference on principles of distributed computing (PODC) [30].

Definition Let us consider a system of processes, each with its own unique identity. In such a context, an algorithm is *symmetric* if the only way to distinguish processes is by comparing their identifiers (names). This means that the set of the process identifiers defines a specific data type such that the identifiers can be written, read, and compared, but there is no way to "look inside" an identifier, which means that other operations cannot manipulate process identifiers. Thus, identifiers cannot be used to index the entries of a shared array.

Two types of symmetric algorithms can be defined according to how much information can be derived from comparing two identifiers [30, 31]. The family of *symmetric algorithms with arbitrary comparisons* allows the three operations $=$, $>$, and $<$ to be applied on process identifiers. Thus, symmetry with arbitrary comparisons allows to totally order the processes according to their identifiers. The family of *symmetric algorithms with only equality* is more restrictive. It allows only the operation $=$ to be applied to process identifiers (in this case, there is no notion of order on process identifiers).

Process symmetry is important for several reasons. One is related to the fact that, while the size of the possible process name-space can be huge (e.g., 2^{32}), the number of processes n is usually relatively much smaller (e.g., $n = 100$). Such a very large name-space does not allow process identifiers to be used as an index to access shared registers. There are two ways to address this issue. One consists in using a process renaming algorithm which allows the size of the name-space to be reduced to n in failure-free systems (this requires an algorithm that needs $2\lceil \log n \rceil + 1$ atomic read/write registers [30]), and to $2n - 1$ in asynchronous systems where processes may crash (see [4, 9]). Another option consists in designing symmetric algorithms. It is important to observe that symmetry with equality means "egalitarian" because a process cannot use its identifier to obtain specific rights. In this sense, process symmetry with equality is the "last" step before process anonymity.

The RW communication model This model is the most basic communication model, namely the only way for processes to communicate is by reading and writing atomic shared registers [16, 19, 27, 31] (the shared registers are the cells of a distributed Turing machine which allows processes to cooperate). A register that

can be written and read by any process is a multi-writer multi-reader (MWMR) register. If a register can be written by a single (predefined) process and read by all, it is a single-writer multi-reader (SWMR) register.

As already indicated, due to the definition of process symmetry, process identifiers cannot be used as pointers to index shared registers. Consequently, process symmetry requires that the shared read/write registers be MWMR registers.

Illustration: Symmetric deadlock-free mutex in RW systems To illustrate process symmetry, let us consider Algorithm 1. This algorithm, due to Styer and Peterson [30], is a process symmetry with equality mutual exclusion algorithm for n processes, where communication is through MWMR atomic registers (RW communication model).

```
operation acquire() is    % invoked by process p
(1)   repeat
(2)     wait(TURN = ⊥); TURN ← p;
(3)     repeat
(4)       for each k ∈ {1, ..., n − 1} do
(5)           if (LOCK[k] = ⊥) then LOCK[k] ← p end if end for;
(6)       locked_p ← ∧_{1≤k≤n−1}(LOCK[k] = p)
(7)     until TURN ≠ p ∨ locked_p end repeat;
(8)     if (TURN = p)
(9)       then return()
(10)      else for each k ∈ {1, ..., n − 1} do
(11)           if (LOCK[k] = p) then LOCK[k] ← ⊥ end if end for
(12)    end if
(13)  end repeat.

operation release() is    % invoked by process p
(14)  TURN ← ⊥;
(15)  for each k ∈ {1, ..., n − 1} do
(16)    if (LOCK[k] = p) then LOCK[k] ← ⊥ end if end for;
(17)  return().
```

Algorithm 1: Symmetric with equality mutex in RW systems [30]

The processes have distinct identities p, q, etc., which can be compared only for equality; $\perp$ is a default process identity (different from the processes' identities). The processes communicate through atomic read/write registers. *TURN* is an atomic MWMR register initialized to $\perp$. Then, it may contain the identity of a

process competing for the critical section. $LOCK[1..n-1]$ is an array of atomic MWMR registers, initialized to $\perp$. The entries of $LOCK$ can be seen as locks that a process needs to capture (by writing its identity in each of them) to enter the critical section. Finally, each process p has a local variable denoted $locked_p$, whose initial value is irrelevant.

To enter the critical section, process p invokes acquire(), which consists of a repeat loop (lines 1-13) that p will exit when it executes the return() statement (line 9). Process p first waits until $TURN = \perp$; when this occurs, it writes its identity in $TURN$ (line 2). Then, it enters an internal repeat loop (lines 3-7). Inside this loop, process p strives to deposit its identity in as many free locks as possible (lines 4-5). When this is done, process p computes if –from its asynchronous and local point of view– it has captured all the locks (assignment to $locked_p$ at line 6). If $TURN \neq p$ or $locked_p =$ `true`, process p exits the internal loop. If $TURN = p$ (in this case $TURN$ has not been modified since p wrote its identity in it), p enters the critical section (lines 8-9). In the other case, before re-entering the main repeat loop, p resets to their initial values all the locks it has previously acquired (lines 10-11).

To exit the critical section, process p invokes the operation release(), which resets to their initial values the shared register $TURN$ and all the locks containing its identity (lines 14-16).

It is easy to see that this algorithm uses exactly n atomic read/write registers and is memoryless (a new invocation of acquire() by a process does not use information on its previous invocations). Moreover, the size of each shared register is bounded by $\log(n+1)$. Proofs of this algorithm can be found in [30, 31].

Remark It is interesting to notice that, while mutual exclusion cannot be solved in the RW communication model when the processes have no identifiers (i.e., are anonymous), it can be solved with process symmetry, which, as already mentioned, is the last step before process anonymity.

3 Process Anonymity

Definition For privacy reasons, some applications must hide the identities of the processes involved. On another side, some applications (e.g., sensor networks) are made up of tiny computing entities with no identifier. This defines the process anonymous computing model, which is characterized by the fact that there is no way for a computing entity (process) to be distinguished from another computing entity. In such a model, not only do the processes have no identities, but they have the same code and the same initialization of their local variables (otherwise, some processes could be distinguished from the others). As for process symmetry, the

Symmetric with equality mutual exclusion on top of shared read/write registers was addressed and solved for the first time in 1989 [30]. This article proves the following lower bounds results and presents associated optimal algorithms.

- n shared read/write registers are necessary and sufficient for deadlock-free symmetric mutual exclusion for n processes.
- $(2n - 1)$ shared read/write registers are necessary and sufficient for memoryless starvation-free symmetric mutual exclusion. "Starvation-free" that that any process that tries to enter the critical section eventually enters it. "Memoryless" means that a process that tries to enter the critical section does not use any information about its previous attempts to enter the critical section.

A symmetric with equality leader election algorithm, in which all the processes are required to participate, was also presented in [30]. This algorithm requires three shared read/write registers, which was conjectured to be necessary. It has recently been shown that a single shared read/write register is sufficient [14].

Sidebar 1: Symmetric mutual exclusion and election in RW systems

notion of SWMR is meaningless for process anonymity: any process may apply any operation to any register.

Process-anonymous failure-free shared memory systems have been studied in [5] where (assuming each process knows the number of processes n) a characterization is presented of problems solvable despite process-anonymity. Relations between the broadcast communication abstraction and reliable process-anonymous shared memory systems have been studied in [3]. Anonymous failure-prone shared-memory systems have been studied in [15], where an answer is presented to the question "What can be deterministically implemented in the process-anonymous crash-prone model?" (deterministically means here that randomized algorithms cannot be used).

Illustration: Obstruction-free binary consensus in asynchronous RW systems To illustrate process-anonymity, let us consider the binary consensus problem in an asynchronous read/write system in which any number of processes may crash. Consensus is one of the most important problems of fault-tolerant distributed computing. Similar to mutual exclusion which is at the core of centralized systems, consensus is at the core of many crash-prone distributed computing problems [24].

Process-anonymous systems have been studied since 1980 in the context of message-passing systems in [2], where several impossibility results are established (e.g., the impossibility to deterministically elect a leader). Characterizations of problems that can be solved in reliable asynchronous message-passing systems despite process anonymity, can be found in [6, 34]. Failure detectors suited to crash-prone asynchronous process-anonymous systems have been introduced and investigated in [7, 8].

Sidebar 2: Process anonymity in message-passing systems

In this problem, each process proposes a value (operation propose()), and must decide on a value. Binary means that only the values 0 and 1 can be proposed. The operation propose() returns the value decided by the invoking process. The following properties define consensus:

- Validity: If a process decides on a value, this value was proposed by a process.
- Agreement: no two processes decide on different values.
- Termination (Wait-freedom): The invocation of propose() by a process that does not crash terminates.

One of the most important results of distributed computing is the impossibility to design a deterministic consensus algorithm satisfying the wait-freedom liveness property [16, 20] in the presence of asynchrony and process crashes (be the communication medium message-passing [11], or RW registers [20]). This impossibility result, established in the context of non-anonymous processes, extends trivially to process-anonymous systems. One way to circumvent this impossibility result is to weaken the termination property as follows (this property, called *obstruction-freedom*, was introduced in [17]).

- Termination (Obstruction-freedom): If process p invokes propose() and all other processes that have pending propose() operations pause during a long enough period, then p terminates its operation.

The notion of "long enough" captures the fact that process p is the only process that continues its execution until it returns from its invocation of propose().

Algorithm 2 described below is due to Rachid Guerraoui and Eric Ruppert [15]. It is a process-anonymous binary consensus in which the processes communicate through MWMR registers, namely a two-dimensional array $SM[0..1, 1..]$ whose second dimension is unbounded. Each register $SM[x, y]$ is initialized to the default value `down`, and it can then take the value `up`. $SM[x, y]$ can be seen as a flag

raised forever by a process when some condition is satisfied. Process p locally manages a current estimate of the decision value $est_p \in \{0, 1\}$, the opposite value denoted $opposite_p$, and an iteration number k_p.

```
operation propose(v) is    % invoked by process p
(1) est_p ← v; k_p ← 0;
(2) repeat
(3)   k_p ← k_p + 1; opposite_p ← 1 − est_p;
(4)   if (SM[opposite_p, k_p] = down)
(5)     then SM[est_p, k_p] ← up;
(6)       if (k_p > 1) ∧ (SM[opposite_p, k_p − 1] = down) then return(est_p) end if
(7)     else est_p ← opposite_p
(8)   end if
(9) end repeat.
```

Algorithm 2: Obstruction-free binary consensus in RW systems [15]

This algorithm can be seen as running a competition between two teams of processes, the team of the processes that champion 0, and the team of the processes that champion 1. Process p first progresses to its next iteration (line 3). Iteration numbers k can be seen as defining a sequence of rounds executed asynchronously by the processes. Hence, the state of the flags $SM[0, k]$ and $SM[1, k]$ (which are up or down) describes the state of the competition at round k. When process p enters round k, there are two cases.

- If the flag associated with this round s (k_p) and the other value ($opposite_p$) is up (i.e., the predicate of line 4 is not satisfied), p changes its mind passing from the group of processes that champion est_p to the group of processes that champion $opposite_p$ (line 7). It then proceeds to the next round.
- If the flag associated with this round and the other value is down (the predicate of line 4 is then satisfied), maybe est_p can be decided. To this end, p indicates first that est_p is competing to be the decided value by raising the round k_p flag $SM[est_p, k_p]$ (line 5). The decision involves the two last rounds, namely $(k_p - 1)$ and k_p, attained by p (hence, the sub-predicate $k_p > 1$ at line 6). If p sees both the flags measuring the progress of $opposite_p$ equal to down at round $(k_p - 1)$ and round k_p (predicate $SM[opposite_p, k_p]$ at line 4, and predicate $SM[opposite_p, k_p - 1]$ at line 6), $opposite_p$ is defeated, and p consequently decides est_p.

  To show this is correct, let us consider the smallest round k during which a process decides. Moreover, let p_i be a process that decides during this round, v the value it decides, and τ the time at which p reads $SM[1 - v, k_p -$

1] before deciding (line 6 of round k). As p decides, at time τ we have $SM[1-v, k_p - 1] = \texttt{down}$. This means that, before time τ, no process changed its mind from v to $1-v$ at line 6. The rest of the proof consists in showing that no process p_j started round k before time τ with $est_j = 1-v$. A proof of this algorithm ensures the consensus is given in [15].

4 Memory Anonymity

Definition Control (processes) and data (memory) are the two pillars of computing. So, while anonymity can be applied to processes, what is the meaning of "memory anonymity"? It means that different processes can have different names for the same register. Let the shared memory be made up of $m \geq 1$ atomic registers $AM[1..m]$. While in a non-anonymous memory $AM[x]$ denotes the same register for all the processes, in an anonymous memory $AM[x]$ can denote some register for process p and a different register for another process q. So, there is an addressing disagreement on the names used by the processes to access the registers. More precisely, an anonymous memory $AM[1..m]$ is such that:

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

The notion of anonymous shared memory has been recently introduced in [32]. The work in [32] was inspired by Michael O. Rabin's paper on solving the choice coordination problem [23].

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

identifiers for an external observer	identifiers for process p	identifiers for process q
$AM[1]$	$AM_p[2]$	$AM_q[3]$
$AM[2]$	$AM_p[3]$	$AM_q[1]$
$AM[3]$	$AM_p[1]$	$AM_q[2]$
permutation	$f_p() : [2, 3, 1]$	$f_q() : [3, 1, 2]$

Table 1: An illustration of an anonymous memory model

Motivating anonymous shared memory Anonymous shared memory have two main 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 both anonymous processes and anonymous memories, and providing associated necessary and sufficient conditions, helps us determine the weakest system assumptions under which the fundamental election problem can be solved.

The second motivation is application-oriented. In [25, 26], it is shown 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.

The authors model histone modifiers (which are a specific type of proteins) as two different types of writer processors and two different types of eraser processors that communicate by accessing an *anonymous* shared memory array which corresponds to a stretch of DNA, and for such a setting formally define the epigenetic consensus problem.

Thus, anonymous shared memories are useful in biologically inspired distributed systems [21, 22], and mastering fundamental distributed computing problems in such an adversarial context could reveal to be important from an application point of view. The similarities and differences between distributed computations in biological and computational (shared-memory and message-passing) systems are explored in [21, 22].

The RMW communication model In addition to the basic RW communication model used previously, we also consider a second communication model denoted RMW (Read-Modify-Write). This model is the RW communication model enriched with the operation Compare&Swap(), which is an atomic conditional write. More precisely, when process p invokes Compare&Swap($AM_p[x], old, new$), where *old* and *new* are two values, it atomically assigns the value *new* to $AM_p[x]$ and returns `true` if $AM_p[x] = old$. Otherwise, $AM_p[x]$ is not modified, and the value `false` is returned.

Necessary and sufficient conditions for mutual exclusion and election in symmetric processes and anonymous memory systems Considering a failure-free system where processes are not anonymous, but the memory is anonymous, the following necessary and sufficient conditions relate the number n of processes and the size m of the anonymous memory to solve two classical problems that are

mutual exclusion and election. These conditions capture the minimal information about the pair $\langle n, m\rangle$ needed to break the symmetry that allows these problems to be solved despite memory anonymity. Let $M(n)$ be the set of the positive integers which are relatively prime with the integers $2, \ldots, n$, i.e., $M(n) = \{m : \forall\, \ell \in \{2, ..., n\} : \mathsf{gcd}(\ell, m) = 1\}$, and let $M'(n) = M(n) \setminus \{1\}$.

- Mutual exclusion can be solved by a symmetric algorithm in a system made up of n (non-anonymous) processes communicating through an anonymous memory of size m accessed by RMW (resp. RW) operations if and only if $m \in M(n)$ (resp. $m \in M'(n)$), $m \in M(n)$ (resp. $m \in M'(n)$). The upper bound was established in [1] and the lower bound in [32].
- Leader election can be solved by a symmetric algorithm, in which all the processes are required to participate, in the systems that consist of of n (non-anonymous) processes communicating through an anonymous memory of size m accessed by RMW or RW operations if and only if $\mathsf{gcd}(m, n) = 1$ [14].

Let us observe that while the RMW operations allow mutual exclusion to be solved in more cases than the RW operations alone, this is no longer true for leader election. Let us also observe that while the conditions $m \in M(n)$ and $m \in M'(n)$ are at the heart of their proofs, they do not appear explicitly in the algorithms.

Illustration: Symmetric processes anonymous memory deadlock-free mutex in RMW systems Algorithm 3 (from [1]) is a symmetric with equality anonymous memory deadlock-free mutual exclusion algorithm. It is based on RMW communication operations and assumes $m \in M'(n)$. The registers of the memory are initialized to $\perp$, and p denotes the identifier of the process that invokes the acquire() or release() operation. Let a register be *free* if it contains $\perp$. We say that a register $AM_p[x]$ is *owned* by process p if $AM_p[x] = p$.

When it invokes acquire(), process p enters a repeat loop in which it first tries to own as many registers as possible by writing its identity in as many free registers as possible (line 2). Then p a reads all the registers (line 3) and computes how many registers –from its local and asynchronous point of view– contains the identity that appears the most frequently ($most\_present_p$, line 4) and how many registers it owns ($owned_p$, line 5). If $owned_p < most\_present_p$, process p momentarily withdraws from the competition, resetting to $\perp$ the registers it owns (line 7), until it sees all registers equal to $\perp$ (lines 8-10). If $owned_p \geq most\_present_p$, continues competing until it owns a majority of registers (line 12). When this occurs, p enters the critical section. When it invokes release() process p resets all the registers it owns to their initial value (line 13. A proof of this algorithm can be found in [1].

```
operation acquire() is
(1)  repeat
(2)    for each x ∈ {1, ..., m} do Compare&Swap(AM_p[x], ⊥, p) end for;
(3)    for each x ∈ {1, ..., m} do view_p[x] ← AM_p[x] end for;
(4)    most_present_p ←
           maximum number of times the same non-⊥ value appears in view_p;
(5)    owned_p ← (|{x ∈ {1, ..., m} : view_p[x] = p}|);
(6)    if owned_p < most_present_p then
(7)      for each x ∈ {1, ..., m} do
             if (view_p[x] = p) then AM_p[x] ← ⊥ end if end for;
(8)      repeat
(9)        for each x ∈ {1, ..., m} do view_p[x] ← AM_p[x] end for
(10)     until ∀ x ∈ {1, ..., m} : view_p[x] = ⊥ end repeat
(11)   end if
(12) until owned_p > m/2 end repeat.

operation release() is
(13) for each x ∈ {1, ..., m} do Compare&Swap(AM_p[x], p, ⊥) end for.
```

Algorithm 3: Symmetric mem.-anony. deadlock-free mutex in RMW systems [1]

It is interesting to note that it is not known whether a symmetric memory-anonymous *starvation-free* mutual exclusion exists. This constitutes a challenging research problem.

Illustration: Symmetric processes memory-anonymous consensus As far as consensus is concerned, a process symmetry with equality obstruction-free consensus algorithm for $n \geq 1$ and $m \geq 2n - 1$ anonymous RW registers is presented in [32].

Illustration: Symmetric processes memory-anonymous election A symmetric algorithm electing a leader in a system where the processes communicate through RW registers is described in [14]. As shown in [13], such an algorithm can be used to de-anonymize an anonymous memory. This election algorithm requires that all the processes participate in the algorithm. It assumes that $\mathsf{gcd}(m, n) = 1$, which is shown to be necessary and sufficient for the election of a single leader. If up to d leaders can be elected the condition becomes $\mathsf{gcd}(m, n) \leq d$ (the number of elected leaders is then ℓ such that $1 \leq \ell \leq d$).

5 Full Anonymity

The ultimate question is now: Are there problems that can be solved when both the processes and the memory are anonymous?

An illustration: Consensus in the RW model As shown in [28], it appears that it is possible to solve obstruction-free consensus in a fully anonymous crash-prone system made up of $n = 2$ processes and $m \geq 3$ anonymous registers, as shown by Algorithm 4.

The anonymous memory is made up of $m \geq 3$ MWMR atomic registers $AM[1..m]$. Each anonymous process p manages a local array $view_p[1..m]$ which will contain a local copy of the anonymous memory, a local variable k_k that is an index to address the entries of $view_p[1..m]$, and a local estimate of the decision value est_p.

```
operation propose(v) is    % invoked by anonymous process p
(1)  est_p ← v;
(2)  repeat
(3)    for each k_p ∈ {1, ..., m} do view_p[k_p] ← AM_p[k_p] end for;
(4)    if (∃ w appearing in a majority of entries of view_p[1..m])
                                              then est_p ← w end if;
(5)    k_p ← arbitrary index j such that view_p[k_p] ≠ est_p if any, otherwise 0;
(6)    if (k_p ≠ 0) then AM_p[k_p] ← est_p end if;
(7)  until view_p[1] = view_p[2] = ··· = view_p[m] = est_p) end repeat;
(8)  return(est_p).
```

Algorithm 4: Fully anony. obst.-free consensus for 2-process RW systems [28]

When process p invokes propose(v), it first deposits v in est_p (line 1), and enters a repeat loop in which it first scans the anonymous registers (line 3). If it sees a majority value w, it adopts w as its new estimate of the decision value (line 4), and writes it in an anonymous register that storing a different value from w (line 5-6). The process p repeats the previous statements until, after scanning the anonymous memory, all its registers contain the same estimate value, which is then decided. A proof of this algorithm is given in [28].

While obstruction-free consensus can be solved for two processes despite full anonymity, crashes and asynchrony, neither an algorithm nor a necessary and sufficient condition are known for the case $n > 2$. This constitutes a challenging research problem.

It is interesting to note that, while it is possible to solve binary consensus for two processes in a fully anonymous crash-prone system using only 3-valued

registers, this is not possible to do so using only 2-valued registers (i.e., bits). It was recently proved in [33] that there is no obstruction-free consensus algorithm for two non-anonymous processes using only anonymous bits. Thus, as shown in [33], it follows that anonymous bits are strictly weaker than anonymous (and hence also non-anonymous) multi-valued registers.

Illustration: Consensus in the RMW model As consensus with the wait-freedom liveness property cannot be solved in a non-anonymous RW system [11, 20], it cannot be solved either in an anonymous asynchronous crash-prone RW system. This impossibility no longer holds in a crash-prone fully anonymous system where the processes communicate with RMW operations, as shown by Algorithm 5 [28].

```
operation propose(v) is    % invoked by process p
(1)  for each k ∈ {1, 2, ..., m} do Compare&Swap(AM_p[k], ⊥, v) end for;
(2)  max ← max(AM_p[1], ...AM_p[m]);
(3)  return(max).
```

Algorithm 5: Fully anony. obst.-free consensus in RMW systems [28]

This very simple algorithm assumes that the proposed values can be totally ordered and works for any value of n and m. All the registers of the anonymous memory $AM[1..m]$ are initialized to $\perp$. The algorithm is based on a "first write, then read" access pattern. Each process p strives to write the value v it proposes in any order in all the registers. Then it returns the greatest value it reads from the anonymous memory (max is a local variable).

Assuming at least one process does not crash, there is a finite time after which (whatever the concurrency/failures pattern), each anonymous register contains a non-$\perp$ value. Moreover, a greater value cannot erase a smaller value already written in a register. This guarantees that a single value can be decided.

Mutex and election in a fully anonymous system Recent results concern mutex and election in fully anonymous system. On the negative side, none of these problems can solved in systems where the anonymous processes communicate through RW registers. Differently they can be solved when communication is through RMW registers The reader will consult [29] for mutex and [18] for election. (In the election problem where processes are anonymous it is required that when a processes terminates the algorithm it knows if it or not a leader.).

6 Conclusion

The purpose of this article was to be a simple introductory presentation of the notions of process symmetry, process anonymity, and memory anonymity in asynchronous systems where communication is through shared memory. To this end, two fundamental (and practically relevant) problems encountered in concurrent and distributed systems have been considered [24]: mutual exclusion in failure-free systems and consensus in crash-prone systems. The following tables summarize what can be done in this context. They also clearly express the additional computability power provided by the RMW communication model with respect to the RW communication model.

Mutual exclusion	communication	nec. & suf. condition	reference
process symmetry (with eq.)	RW	$n > 1, m \geq n$	[30]
memory anonymity	RW	$n > 1, m \in M'(n)$	[1] UB, [32] LB
memory anonymity	RMW	$n > 1, m \in M(n)$	[1]
full anonymity	RW	impossible	[29]
full anonymity	RMW	$n > 1, m \in M(n)$	[29]

Table 2: Deadlock-free mutual exclusion (asynchronous failure-free systems)

Table 2 concerns mutual exclusion (LB and UB stand for lower bound and upper bound, respectively). Table 3 concerns consensus. Let us remember that n is the number of processes, m is the size of the memory, $M(n) = \{m : \forall\, \ell \in \{2, ..., n\} : \mathsf{gcd}(\ell, m) = 1\}$, and $M'(n) = M(n) \setminus \{1\}$. Also, in the tables, when we write "memory anonymity", we mean that the memory is anonymous and the processes are symmetric.

Consensus	comm.	progress property	sufficient condition	reference
process anonymity	RW	obstruction-freedom	$n > 1, m = \infty$	[15]
memory anonymity	RW	obstruction-freedom	$n > 1, m \geq 2n - 1$	[32]
full anonymity	RW	obstruction-freedom	$n = 2, m \geq 3$	[28]
full anonymity	RMW	wait-freedom	$n > 1, m \geq 1$	[28]

Table 3: Consensus (asynchronous crash-prone systems)

References

[1] 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)

[2] 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)

[3] Aspnes J., Fich F.E., and Ruppert E., Relationship between broadcast and shared memory in reliable anonymous distributed systems. *Distributed Computing*, 18(3):209-219 (2006)

[4] Attiya H., Bar-Noy A., Dolev D., Peleg D., and Reischuk R., Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524-548, 1990.

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

[6] Attiya H., Snir M. and Warmuth M.K., Computing on an anonymous ring. *Journal of the ACM*, 35(4):845-875 (1988)

[7] Bonnet F. and Raynal M., The price of anonymity: optimal consensus despite asynchrony, crash and anonymity. *ACM Transactions on Autonomous and Adaptive Systems*, 6(4), 28 pages (2011)

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

[9] Castañeda A., Rajsbaum S., and Raynal M., The renaming problem in shared memory systems: an introduction. *Computer Science Review*, 5:229-251 (2011)

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

[11] Fischer M.J., Lynch N.A., and Paterson M.S., Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374-382 (1985)

[12] Gardner M., *The ambidextrous universe: mirror asymmetry and time-reversed worlds.* 293 pages, Penguin Books (1982)

[13] Godard E., Imbs D., Raynal M., Taubenfeld G., Leader-based de-anonymization of an anonymous read/write memory. *Theoretical Computer Science*, 836(10):110-123 (2020)

[14] Godard E., Imbs D., Raynal M., Taubenfeld G., From Bezout identity to space-optimal leader election in anonymous memory systems. *Proc. 39th ACM Symposium on Principles of Distributed Computing (PODC'20)*, ACM press, pp. 41-50 (2020)

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

[16] Herlihy M., Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13 (1):124-149 (1991)

[17] Herlihy M.P., Luchangco V., and Moir M., Obstruction-free synchronization: double-ended queues as an example. *Proc. 23th Int'l IEEE Conference on Distributed Computing Systems (ICDCS'03)*, IEEE Press, pp. 522-529 (2003)

[18] Imbs D., Raynal M., and Taubenfeld G., Election in fully anonymous shared memory systems: tight space bounds and algorithms. *Submitted to publication*, LIPIcs, 14 pages (2021)

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

[20] Loui M.C., and Abu-Amara H.H., Memory requirements for agreement among unreliable asynchronous processes. *Parallel and Distributed Computing: Vol. 4 of Advances in Computing Research,* JAI Press, 4:163-183 (1987)

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

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

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

[24] Rajsbaum S. and Raynal M., Mastering concurrent computing through sequential thinking: A half-century evolution. *Communications of the ACM*, Vol. 63(1):78-87 (2020)

[25] 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)

[26] Rashid S., Taubenfeld G. and Bar-Joseph Z., The epigenetic consensus problem. *28th Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO'21)*, Springer LNCS 12810, pp. 146–163 (2021)

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

[28] Raynal M. and Taubenfeld G., Fully anonymous consensus and set agreement algorithms. *Proc. 8th Int'l Conference on Networked Systems (NETYS'20)*, Springer LNCS 12129, pp. 314-328 (2020)

[29] Raynal M. and Taubenfeld G., Mutual exclusion in fully anonymous shared memory systems. *Information Processing Letters*, Vol. 158, 105938, 7 pages (2020)

[30] 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)

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

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

[33] 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)*, LNCS 11639, pp. 293-308 (2019)

[34] 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)