Coordination Without Prior Agreement

Gadi Taubenfeld
The Interdisciplinary Center
P.O. Box 167
Herzliya, Israel 46150
tgadi@idc.ac.il

ABSTRACT

Assuming that there is an a priori agreement between processes on the names of shared memory locations, as done in almost all the publications on shared memory algorithms, is tantamount to assuming that agreement has already been solved at the lower-level. From a theoretical point of view, it is intriguing to figure out how coordination can be achieved without relying on such lower-level agreement. In order to better understand the new model, we have designed new algorithms without relying on such a priori lower-level agreement, and proved space lower bounds and impossibility results for several important problems, such as mutual exclusion, consensus, election and renaming. Using these results, we identify fundamental differences between the standard model where there is a lower-level agreement about the shared register's names and the strictly weaker model where there is no such agreement.

KEYWORDS

Shared memory; memory-anonymous algorithms; unnamed register; anonymous register; mutual exclusion; agreement; renaming.

1 INTRODUCTION

A central issue in distributed systems is how to coordinate the actions of asynchronous processes. In the context where processes communicate via reading and writing from shared memory, in almost all published concurrent algorithms it is assumed that the shared memory locations have global names which are a priori known to all the participating processes. From a theoretical point of view, it is intriguing to figure out what and how coordination can be achieved without relying on such lower-level agreement about the names of the memory locations.

We assume that all inter-process communications are via shared registers which are initially in a known state. Access to the registers is via atomic read and write operations. However, from the point of view of the processes, the registers *do not have global names*: the first register examined and the subsequent order in which registers are scanned may be different for each process. That is, a single register may be considered the fifth register by one process and the eighth by another. Even the order of the names may be different. Thus, for example, one process may scan four registers in order 3, 2, 1, 4 while another scans 2, 4, 1, 3.

We call algorithms that are correct for a model where the registers do not have global names, *memory-anonymous* algorithms. The lack of global names makes it convenient to think of each process as being assigned an initial register and an ordering of the registers which determines how it scans the registers. Our interest in such a weak computational model is of a theoretical nature; however, the *plasticity* of memory-anonymous algorithms – their ability to operate for any assigned ordering of the registers – may be found useful in practice. When using such algorithms, specific ordering can be assigned for reducing memory contention which may help in improving performance. Furthermore, various optimizations enable reordering memory references as it allows much better performance. Because of its plasticity, memory-anonymous algorithms may need to use only a small number of memory barriers to prevent reordering.

In order to better understand the new model, we have designed new memory-anonymous algorithms, proved space lower bounds and impossibility results for several important problems, which include mutual exclusion, consensus, election and renaming. Using these results, we identify fundamental differences between the standard model where there is a lower-level agreement about shared register's names and the *strictly weaker* model where there is no such agreement.

2 PRELIMINARIES

Our model of computation consists of a fully asynchronous collection of n processes which communicate via atomic read/write registers that do not have global names. Thus, algorithms should be correct assuming a very powerful adversary, which can determine (essentially) the order in which processes access the registers.

The fact that the registers do not have global names implies that only multi-writer multi-reader (MWMR) registers are possible. A MWMR register can both be written and read by all the processes. With an *atomic* register, it is assumed that operations on the register occur in some definite order. That is, reading or writing an atomic register is an indivisible action. In the sequel, by *registers* we mean *atomic* registers. Asynchrony means that there is no assumption on the relative speeds of the processes. Each process has a unique identifier which is a positive integer. It is *not* assumed that the identifiers are taken from the set $\{1, ..., n\}$. Thus, a process does not a priori know the identifiers of the other processes.

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 any

PODC'17, July 25-27, 2017, Washington, DC, USA

DOI: http://dx.doi.org/10.1145/3087801.3087807

identifier. Thus, it is not possible to know whether an identifier is odd or even. Two variants of symmetric algorithms can be defined:

- *Symmetric with equality*, means that a process can determine if two identifiers are the same or not, but if they are different then nothing else can be determined.
- *Symmetric with arbitrary comparisons*, means that a process can learn something about identifiers if two unequal identifiers are compared. For example, comparisons can be defined that depend on a total order.

We assume that there is *no a priori agreement* between the processes regarding the *order* of their identifiers. Thus, throughout the paper, we assume that only comparisons for equality are allowed. Furthermore, a process will only compare its own identifier with another, and not compare it with a constant value.

In some cases, we will assume that processes may fail by crashing; that is, they fail only by never entering the algorithm or by leaving the algorithm at some point and thereafter permanently refraining from writing the shared registers. A process that crashes is said to be *faulty*, otherwise it is *correct*.

Several progress conditions have been proposed for algorithms in which processes may fail. *Wait-freedom* guarantees that *every* active process will always be able to complete its pending operations in a finite number of its own steps [14]. *Obstruction-freedom* guarantees that an active process will be able to complete its pending operations in a finite number of its own steps, if all the other processes "hold still" long enough [15]. Obstruction-freedom does not guarantee progress under contention.

In a model where participation is required, every correct process must eventually become active and execute its code. A more common and practical situation is one in which participation is not required. Throughout the paper, we will assume that participation is *not* required.

3 MEMORY-ANONYMOUS MUTUAL EXCLUSION

None of the published mutual exclusion algorithms (that I know of) using registers is memory-anonymous. Below we present the first memory-anonymous mutual exclusion algorithm for two processes.

3.1 The problem

The mutual exclusion problem is to design an algorithm that guarantees mutually exclusive access to a critical section among a number of competing processes [Dij65]. It is assumed that each process is executing a sequence of instructions in an infinite loop. The instructions are divided into four continuous sections: the remainder, entry, critical and exit. The exit section is required to be wait-free – its execution must always terminate. It is assumed that processes do not fail, and that a process always leaves its critical section. The *mutual exclusion problem* is to write the code for the entry and the exit sections in such a way that the following *two* basic requirements are satisfied.

- *Deadlock-freedom:* If a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section.
- *Mutual exclusion:* No two processes are in their critical sections at the same time.

Satisfaction of the above two properties is the minimum required for a mutual exclusion algorithm.

3.2 A result for two processes and its implications

We prove the following result:

THEOREM 3.1. *There is a memory-anonymous symmetric deadlock-free mutual exclusion algorithm for two processes using $m \geq 2$ registers if and only if m is odd.*

In the standard model, where *there is* a lower-level a priori agreement regarding the registers names, the following three general properties hold:

- If a problem has a solution using ℓ registers then it also has a solution using m registers, for every $m \geq \ell$. (Simply ignore $m-\ell$ of the registers. This requires a prior agreement on which $m - \ell$ registers should be ignored.)
- For every $n \geq 2$, if a problem has a solution for n processes, with distinct identifiers taken from the set $\{1, ..., n\}$, using registers then it has a solution (not necessarily symmetric) using n registers. (This follows from the fact that multi-writer registers can be wait-free implemented from single-writer registers.)
- For every $n \geq 2$, there is a symmetric mutual exclusion algorithm for n processes using n registers, which uses only comparisons for equality. (See [23].)

It follows immediately from Theorem 3.1, that these three properties do *not* hold for a model where there is *no* lower-level agreement regarding the registers names assuming comparisons for equality.

3.3 A memory-anonymous mutual exclusion algorithm

We present below a memory-anonymous deadlock-free mutual exclusion algorithm for two processes using m registers, for any odd number of registers $m \geq 3$. The question whether there is an algorithm for more than two processes is open (at first glance this open problem looks simple, but it is not).

Each participating process scans the m shared registers trying to write its identifier into each one of the m registers. While accessing a register, a process may find that another process has already "visited" that register. In such a case, the process does not write into that register and continues to the next one. Once a process completes scanning the m registers and possibly writing its identifier to some of them, it scans the registers again, this time only reading their values. If it finds that its identifier is written in all the m registers, it safely enters its critical section. If its identifer is written in less than $\lceil m/2 \rceil$ registers, it gives up and sets the registers in which its name is written back to their initial values. If its identifer is written in at least $\lceil m/2 \rceil$ registers (but not in all), it starts all over again. On exiting its critical section, a process sets all the registers back to their initial values.

As the m registers do not have global names, each process independently numbers them. We use the notation $p.i[j]$ to denote the j^{th} register according to process i numbering, for $1 \leq j \leq m$. Recall, that a process' identifier is a positive integer. The code of the algorithm appears in Figure 1.

Code of process i

Constants:
m: an odd integer greater than 2 //m is the # of shared registers
Shared variables:
$p.i[1..m]$: array of m atomic registers, initially all 0 //$m \geq 3$ is odd
Local variables:
$myview[1..m]$: array of m variables
j: ranges over $\{1, ..., n\}$

```
1  repeat                                                                  //begin entry code
2      for j = 1 to m do if p.i[j] = 0 then p.i[j] = i fi od               //scan and write
3      for j = 1 to m do myview[j] := p.i[j] od                            //read the shared array
4      if i appears in less than ⌈m/2⌉ of the entries of myview[1..m] then //lose
5          for j = 1 to m do if p.i[j] = i then p.i[j] = 0 fi od           //clean up
6          repeat                                                          //wait for CS to be released
7              for j = 1 to m do myview[j] := p.i[j] od                    //read the shared array
8          until ∀j ∈ {1, ..., m} : myview[j] = 0                          //try again
9      fi
10 until ∀j ∈ {1, ..., m} : myview[j] = i                                  //end entry code
11 critical section
12 for j = 1 to m do p.i[j] = 0 od                                         //exit code
```

Figure 1: A memory-anonymous symmetric deadlock-free mutual exclusion algorithm for two processes

In the first for loop (lines 2) process i scans the m registers trying to set them all to its identifier. If it does not succeed to set at least $\lceil m/2 \rceil$, it resets the registers with its identifier back to 0 (line 5) and then waits for the critical section to be released (lines 6–8). If it finds out that its identifier is written in all the m registers, it safely enters its critical section (line 10). Otherwise, it go back to the beginning of the repeat loop (line 1) and starts all over again. In its exit code, process i resets all the registers back to 0 (line 12).

3.4 Correctness proof

Below we present a correctness proof for the algorithm.

Theorem 3.2. *The algorithm satisfies mutual exclusion.*

Proof. Let i and j be the identifiers of the two processes. Assume that process i is in its critical section, while process i is in its entry code. Before i enters its critical section the expression in line 10, $\forall j \in \{1, ..., m\} : myview[j] = i$ must be evaluated to true. This means that, before i has entered its critical section, there is a single point in time where the value of each one of the $m \geq 3$ shared registers equals i. After that point, process j might write once into one of the registers overwriting the i value. Thus, process j, when executing line 4, will find that its identifier, j, appears in less that $\lceil m/2 \rceil$ of the entries of the shared array (actually, the value j may appear in at most one entry) and will change back to 0 the single entry in which its identifier may appears (line 5). From that point on, as long as i is in its critical section, the value i will appear in at least $m - 1$ entries of the shared array, preventing j for entering its critical section. □

Theorem 3.3. *The algorithm is deadlock-free.*

Proof. We show that if a process is trying to enter its critical section, then some process eventually enters its critical section. In the first for loop (lines 2) each process scans the m registers trying to set those that are 0 to its identifier. If the process is running alone it will clearly succeed to set them all to its identifier and will enter its critical section. When there is contention (i.e., both processes are in their entry code) since $m \geq 3$ is an odd number, exactly one of the two processes must find that less than $\lceil m/2 \rceil$ of the registers are set to its identifier, will give up, wait in the inner repeat loop (lines 6–8), enabling the other process to write all the m registers and to enter its critical section. In its exit code, a process resets all the registers back to 0 enabling a possibly waiting process to continue. Thus, it is not possible for both processes to simultaneously be in their entry codes forever. □

3.5 A general space lower bound

The result that there is a memory-anonymous symmetric deadlock-free mutual exclusion algorithm for two processes using $m \geq 2$ registers only if m is odd is a special case of the following more general result. Two integers x and y are said to be *relatively prime* if their greatest common divisor is 1. We notice that a number is *not* relatively prime to itself.

Theorem 3.4. *There is a memory-anonymous symmetric deadlock-free mutual exclusion algorithm for n processes using $m \geq 2$ registers only if for every positive integer $1 < \ell \leq n$, m and ℓ are relatively prime.*

Proof. We assume to the contrary that there is a memory-anonymous symmetric deadlock-free mutual exclusion algorithm for n processes using $m \geq 2$ registers such that for some positive integer $1 < \ell \leq n$, m and ℓ are not relatively prime. This means that there is a number $1 < \ell \leq m$ such that ℓ divides m.

The lack of global names for the registers makes it convenient to think of each process as being assigned an initial register and an ordering of the registers which determines how it scans the registers when it runs alone. For example, if a process scans three registers, named 3,4 and 5, in the order 5445334 then the ordering would be 543, omitting duplications.

We arrange the registers as a unidirectional ring of size m, where the direction of the unidirectional ring corresponds to the direction in which the processes scan the registers. So, we pick up ℓ processes, and assign these ℓ processes the same ring ordering, though potentially different initial registers. That is, the processes scan the registers by "walking" around the ring in the same direction. The distance between two registers on the ring is the smallest number of registers between them plus 1. For each process, we assign an initial register such that the distance between any two neighbouring initial registers is exactly m/l. (Two initial registers are neighbours when there is no other initial register between them.)

We run the ℓ processes in lock steps. We first let each one of them take one step (in some order), and then let each one of them takes another step, and so on. Since only comparisons for equality are allowed, processes that take the same number of steps will be at the same state, and thus it is not possible to break symmetry. Thus, either all the processes will enter their critical sections at the same time violating mutual exclusion, or no process will ever enter its critical section violating deadlock-freedom. A contradiction. □

4 A MEMORY-ANONYMOUS OBSTRUCTION-FREE CONSENSUS ALGORITHM

The multi-valued consensus problem is to design an algorithm in which all non-faulty processes reach a common decision based on their initial opinions. It is impossible to solve consensus with a single crash failure using registers only (even in the classical model) [11, 16]. This implies that there is no wait-free consensus algorithm using registers. Below we focus on the design of an obstruction-free consensus algorithm.

The consensus problem is defined as follows: there are n processes, and each process has some input value. The requirements of the consensus problem are that there exists a *decision value* v such that: (1) *Agreement*: all non-faulty process that eventually decide, decide on the same value v, and (2) *Validity*: v is the input value of at least one of the processes. We recall that obstruction-freedom requires that each process that runs alone, for sufficiently long time, must eventually decide.

When there are only two possible input values, the problem is called *binary* consensus. We solve the general *multi-valued* version of the problem. Our algorithm uses a key idea from an earlier algorithm, which uses single-writer registers and snapshot objects (and hence is not memory-anonymous) [5].

4.1 The consensus algorithm

The algorithm uses an array of $2n-1$ shared registers, where each register is defined as a record which has two fields: an *id* field that can store a process identifier and a *val* field which stores the preference of a process. Each participating process scans the $2n-1$ shared registers trying to write its identifier and preference into each one of the $2n-1$ registers. Before each write, the process scans the shared array and operates as follows: if its identifier and preference appears in all the $2n-1$ registers, it decides on its preference, and terminates; otherwise, if some preference appears in at least n of the value fields, the process adopts this preference as its new value.

Afterwards, the process finds some arbitrary entry in the shared array that does not contain both its identifier and preference and writes the pair containing its identifier and preference into that entry. Once the process finishes writing it repeats the above steps until its identifier and preference appear in all the $2n-1$ registers, at which point it can decide on its current preference and terminate.

As in the previous section, we use the notation $p.i[j]$ to denote the j^{th} register according to process i numbering, for $1 \leq j \leq 2n-1$. The code of the algorithm appears in Figure 2.

After process i sets its initial preference to its input (line 1), it executes the repeat loop (lines 2–8) until its identifier and preference appears in all the $2n-1$ registers. Within the repeat loop, it first copies the values of the shared array into the local array $myview[1..2n-1]$ (line 3). If some value appears in at least n of the *val* fields of the entries of $myview[1..2n-1]$ (line 4), the process adopts this value as its preference, by writing this value into *mypref* (line 5). Then, the process finds some arbitrary entry in $myview[1..2n-1]$ that does not contain the pair $(i, mypref)$ (line 6) and writes this pair into that entry (line 7). Clearly, such an arbitrary entry exists. Once the process exits the repeat loop, it decides on the value of *mypref* and terminates.

Remark: Defining each register as a record which has two fields with *global* names, is done only for convenience. The two values in these fields can be encoded as a single value, removing the need for using more than one field. A similar comment applies also for the renaming algorithm presented in the next section.

A note on obstruction-free election: In the election problem, each participating process that terminates, outputs the identifier of some participating process before terminating. It is required that all the participants which terminate output the same identifier, which identifies the elected leader. Obstruction-freedom requires that each process that runs alone, for sufficiently long time, must eventually output a value and terminate. In asynchronous systems where processes communicate using registers, election is impossible with one faulty process even with registers which have global names [11, 19, 26].[1] It is straightforward to use the above consensus algorithm for constructing a memory-anonymous symmetric obstruction-free election algorithm: Each process simply uses its own identifier as its initial input.

4.2 Correctness proof

Below we present a correctness proof for the consensus algorithm.

Theorem 4.1 (agreement under obstruction-freedom). *Every participating process, that runs alone for sufficiently long time, eventually decides; and all the participating processes that decide, decide on the same value and terminate.*

[1] An algorithm that tolerates a single failure does not necessarily satisfies obstruction-freedom, and vice versa.

CODE OF PROCESS i WITH INPUT in_i

Shared variables:
$p.i[1..2n-1]$: array of $2n-1$ atomic registers, initially all fields are 0

Local variables:
$myview[1..2n-1]$: array of $2n-1$ variables, each entry has two fields id and val.
$mypref$: integer ; j: ranges over $\{1, ..., 2n-1\}$

```
1  mypref := in_i
2  repeat
3      for j = 1 to 2n − 1 do myview[j] := p.i[j] od                    //read the shared array
4      if ∃ value ≠ 0 which appears in at least n of the val fields of the entries of myview[1..2n − 1]
5          then mypref := value fi                                      //update preference
6      j := an arbitrary index k ∈ {1, ..., 2n − 1} such that myview[k] ≠ (i, mypref)   //search
7      p.i[j] := (i, mypref)                                            // write
8  until ∀j ∈ {1, ..., 2n − 1} : myview[j] = (i, mypref)                //my id appears everywhere
9  decide(mypref)                                                       //decide
```

Figure 2: A memory-anonymous symmetric obstruction-free consensus algorithm

PROOF. Let process i be the first process to decide, and denote the value that i decides on by v. This means that, before deciding, process i has found that the value of each one of the $2n-1$ shared registers equals (i, v). Each one of the other $n-1$ processes might write into one of the registers overwriting the (i, v) value. Thus, *all* the other processes, when executing line 4, will find that v appears in at least n of the val fields of the entries of $myview[1..2n-1]$, and each one of them will change its preference to v (line 5). From that point on, the only possible decision value is v. Next, we show that each process eventually decides (and terminates) under obstruction-freedom (that is, if it runs alone for sufficiently long time). When a process, say process j, runs alone from some point on in a computation, j will read the shared array (line 3) and set its preference to v (if it is not v already). From that point on, in each iteration of the repeat loop, process j will set one entry of the shared array to (j, v). Thus, after at most $2n-1$ iterations the values of all the $2n-1$ entries will equal (j, v), and process j will be able to exit the repeat loop, decide v and terminate. □

THEOREM 4.2 (VALIDITY). *The decision value is the input of a participating process.*

PROOF. At each point the current preference of a process is either its initial input or a value (different from 0), it has read from the val field of a shared register. Since a process may only write its preference into the val field of a shared registers, the result follows. □

5 A MEMORY-ANONYMOUS OBSTRUCTION-FREE PERFECT RENAMING ALGORITHM

A *renaming* algorithm allows processes with initially distinct names from a large name space to acquire distinct new names from a small name space. A *perfect* renaming algorithm allows n processes with initially distinct names from a large name space to acquire distinct new names from the set $\{1, ...n\}$. A perfect renaming algorithm is *adaptive* if, for any $1 \leq k \leq n$, when only k processes participate, they acquire distinct new names from the set $\{1, ...k\}$. In asynchronous systems where processes communicate using atomic registers there is no perfect renaming algorithm which can tolerate a single failure even with registers which have global names [1, 19, 26]. We show that there is a memory-anonymous obstruction-free adaptive perfect renaming algorithm using registers.

It is straightforward to solve perfect renaming in a model where *there is* an a priori agreement on the names of the registers, given that there is a solution for the election problem. This is done as follows: $n-1$ (obstruction-free) election objects are used. The election objects are indexed 1,2,....,$n-1$. Each process scans the objects, in order, starting with object number 1. At each step, the process applies the election operation, and either: moves to the next object if it is not elected in object $i < n-1$, stops if it is being elected, or stops if it not elected in object $n-1$. The process is assigned either the name equal to the index of the object on which its election operation has succeeded, or n if it is not elected in all $n-1$ objects. This trivial solution requires a priori agreement on an ordering for the election objects, and hence would not work in a model where there is *no* a priori agreement on the registers names.

5.1 The perfect renaming algorithm

The basic idea, as in the above trivial solution, is for each process to participate in up to n election "games", and to acquire a name according to the game in which it is elected. However, the solution does not require a priori agreement on an ordering for the election games. This is achieved by playing each one of these games in the same shared space. The algorithm uses an array of $2n-1$ shared registers, where each register is defined as a record which has *four* fields: an id and val fields, each can store a process identifier, a *round* field which can store a number between 1 to n, and a *history* field which stores a set of pairs of the form (*identifier, value*) where $value \in \{1, ..., n\}$.

CODE OF PROCESS i

Shared variables:

$p.i[1..2n-1]$: array of $2n-1$ registers, initially the fields *id*, *val*, *round*, and *history* are 0,0,0 and $\emptyset$, resp.

Local variables:

$myview[1..2n-1]$: array of $2n-1$ variables, each entry has four fields *id*, *val*, *round*, *history*
$mypref$: integer ; j: ranges over $\{1, ..., 2n-1\}$
$myround$, $mytemp$: ranges over $\{1, ..., n\}$, initially 1
$myhistory$: a set of pairs of the form (*identifier, value*) where $value \in \{1, ..., n\}$, initially $\emptyset$

```
1  repeat
2      mypref := i                                                    //i tries to win in the current round
3      repeat                                                  //in each round one process wins and gets a new name
4          for j = 1 to 2n − 1 do myview[j] := p.i[j] od                          //read the shared array
5          if ∃ j ∃ v : (i, v) ∈ myview[j].history                              //i already got new name!
6              then return(v) fi                                        //return new name and terminate
7          mytemp := max_{j∈{1,...,2n−1}} myview[j].round                  //finding the maximum round #
8          if mytemp > myround then                                                //i is lagging behind
9              j := an arbitrary index k ∈ {1, ..., 2n − 1} such that myview[k].round = mytemp
10             mypref := myview[j].val                                                  //catching up
11             myhistory := myview[j].history                                           //catching up
12             myround := myview[j].round fi                                            //catching up
13         if ∃ v ≠ 0 such that among the entries of myview[1..2n − 1] whose round fields equals myround
               v appears in at least n of the val fields of these entries
14             then mypref := v fi                                          // change preference
15         j := an arbitrary index k ∈ {1, ..., 2n − 1} such that
               myview[k] ≠ (i, mypref, myround, myhistory)                                  //search
16         p.i[j] := (i, mypref, myround, myhistory)                                     //write
17     until ∀j ∈ {1, ..., 2n − 1} : myview[j] = (i, mypref, myround, myhistory)        //success
18     if mypref = i then return(myround) fi                        //return new name and terminate
19     myhistory := myhistory ∪ {(mypref, myround)}                                 //update history
20     myround := myround + 1                                                      //increment round
21 until myround = n                                                    //a single process left
22 return(n)                                                          //last process to acquire a name
```

Figure 3: A memory-anonymous symmetric obstruction-free adaptive perfect renaming algorithm

The algorithm precedes in rounds, where in each round the processes elect one of the participating processes as a leader. These rounds are *local* and do not require global synchronization. In each round, say r, each participating process scans the $2n-1$ registers trying to write its preference, which in this case is its identifier, into each one of the $2n-1$ registers. In addition, the process also writes into each one of the $2n-1$ registers a set which includes the history of all the processes that were elected in previous rounds. Eventually, the processes will agree on some value, say *id*, in round r and *id* will identify the leader for round r. The process whose identity equals *id* will eventually be assigned the value r as its new name. Once a process is elected in round r, it does not participate in later rounds. A process that notices that a leader is elected in around r, updates its history by adding the pair (id, r) to its history, and continues to round $r+1$. After n rounds, all the processes will be assigned new names. As before, we use the notation $p.i[j]$ to denote the j^{th} register according to process i numbering, for $1 \leq j \leq 2n-1$. The code of the algorithm appears in Figure 3.

After process i sets its initial preference to its own identifier (line 2), it executes the inner repeat loop (lines 3–17) until its identifier and preference appears in all the $2n-1$ registers. Within the repeat loop, it first copies the values of the shared array into the local array $myview[1..2n-1]$ (line 4). It then checks if its identifier appears in the history of some process (line 5). If so, it means that it was elected as a leader in one of the previous rounds, in which case it returns the round number in which it was elected as its new name and terminates. Otherwise, if it sees that it is lagging behind (line 8), it is catching up by updating its preference, history and round number (lines 9-12). If some value appears in at least n of the *val* fields of the entries of $myview[1..2n-1]$ whose *round* fields equals *myround* (line 13), the process adopts this value (identifier) as its new preference (line 14). Then, the process finds some arbitrary entry in $myview[1..2n-1]$ that does not contain the tuple $(i, mypref, myround, myhistory)$ (line 15) and writes it into that entry (line 16). Once the process exits the inner repeat loop, it first checks if it was elected in the current round, and if so it

returns the round number and terminates (line 18). Otherwise, it updates its history (line 19) and round number (line 19). In case its round number reaches n, it exits the outer repeat loop, returns n as its new name and terminates. Otherwise, it continues to the next round.

5.2 Correctness proof

Below we present a correctness proof for the renaming algorithm.

Theorem 5.1 (Termination under obstruction-freedom). *Every participating processes, if it runs alone for sufficiently long time, acquires a new name and terminate.*

Proof. Assume that at some point in time some a process, say process i, runs alone. One of the following four scenarios must happen:

(1) Process i checks if its identifier appears in the history of some process (line 5). If so, it returns the round number in which it was elected as its new name and terminates (line 6).
(2) Process i, if needed, updates *once* its *mypref*, *myround* and *myhistory* variables (lines 10, 11, 12, 14). From that point on, in each iteration of the inner repeat loop (lines 3–17), process i sets one entry of the shared array to $(i, mypref, myround, myhistory)$. Thus, after at most $2n-1$ iterations the values of all the $2n-1$ entries equal $(i, mypref, myround, myhistory)$, and process i is able to exit the inner repeat loop. Once process i exits the inner repeat loop, it checks if $mypref = i$, and if so it returns the round number and terminates (line 18).
(3) Process i increments *mypref* by 1 (line 20). In case $mypref = n$, it exits the outer repeat loop, returns n as its new name and terminates.
(4) Process i continues alone to the next round. It first sets *mypref* to i (line 2). From that point on, in each iteration of the inner repeat loop (lines 3–17), process i sets one entry of the shared array to $(i, i, myround, myhistory)$. Thus, after at most $2n-1$ iterations the values of all the $2n-1$ entries equal $(i, i, myround, myhistory)$, and process i is able to exit the inner repeat loop. Once process i exits the inner repeat loop, since $mypref = i$, it returns the current round number as its new name and terminates (line 18).

This completes the proof. □

Theorem 5.2 (Uniqueness). *Every participating processes acquires unique name from the set* $\{1, \ldots n\}$.

Proof. Assume that process i acquires the name r. This means that, at some point in time during round r (before i acquires the name r), for every $j \in \{1, \ldots, 2n-1\}$, $p.i[j].val = i$ and $p.i[j].round = r$. This happens immediately after some process (not necessarily i) has updated a shared register. Each one of the other $n-1$ processes might write into one of the $2n-1$ registers during round r overwriting the value of a register. Thus, *all* the other processes, when executing line 13 during round r, will find that v appears in at least n of the *val* fields of the entries of $myview[1..2n-1]$, and each one of them will change its preference to v during round r (line 14). From that point on, the only possible leader for round r is i. Thus, no other process will be assigned the value r. Since the number assigned to a process corresponds to the round at which it was elected or n if not elected in the first $n-1$ rounds, the assigned name must be from the set $\{1, \ldots n\}$. □

Theorem 5.3 (Adaptivity). *For any* $1 \leq k \leq n$, *when only* k *processes participate, they acquire new names from the set* $\{1, \ldots k\}$.

Proof. If only $k < n$ processes participate, the algorithm will terminate after k rounds. Since the number assigned to a process corresponds to the round at which it was elected (or n if not elected in the first $n-1$ rounds), the assigned name must be from the set $\{1, \ldots k\}$. □

6 IMPOSSIBILITY RESULTS AND SPACE LOWER BOUNDS

We first address the following question: Is a model in which there is no agreement regarding the names of the registers *strictly weaker* than a model in which there is such agreement? The assumption that there is a lower-level a priori agreement regarding an object name can be viewed as an attribute of the object itself. A *named object* is an object for which there is an a priori agreement regarding its name, while an *unnamed object* (also called *anonymous* object) is an object for which there is no agreement regarding its name. Is it possible to implement a named register using unnamed registers? We prove the following result:

Theorem 6.1. *A model in which there is no agreement regarding the names of the registers is strictly weaker than a model in which there is such agreement, even when assuming that processes never fail. Thus, it is not possible to implement a single named register using any number of unnamed registers, even when assuming that processes never fail.*

In the previous sections we have assumed that the number of processes, n, is finite and a priori known. Here we investigate the design of algorithms assuming *no a priori agreement* on the *number* of processes. Our results apply to both the case where the number of processes is finite but not a priori known, and the case where the number of processes is unbounded. We say that that the number of processes is *unbounded* when, in each run, the number of processes that are simultaneously active in any given point in time (i.e., in any state) is finite but can grow without bound.

We prove below three impossibility results for a model where the processes communicated via unnamed registers and there is no a priori known bound on number of processes. These results also hold for algorithms which are *not symmetric*. The results are for deadlock-free mutual exclusion, obstruction-free consensus and obstruction-free adaptive perfect renaming. The impossibility result for deadlock-free mutual exclusion does not hold when using named registers. Thus, the result for deadlock-free mutual exclusion implies Theorem 6.1. The impossibility proofs below are all based on covering arguments and have the same structure.

6.1 Preliminaries

For proving the impossibility results we will use the following notions and notations. An *event* corresponds to an atomic step performed by a process. A (global) *state* of an algorithm is completely

described by the values of the (local and shared) registers and the values of the location counters of all the processes. A *run* is defined as a sequence of alternating states and events (also referred to as steps). For the purpose of the proofs below, it is more convenient to define a run as a sequence of events omitting all the states except the initial state. Since the states in a run are uniquely determined by the events and the initial state, no information is lost by omitting the states.

We will use x, y and z to denote runs. When x is a prefix of y (and y is an *extension* of x), we denote by $(y - x)$ the suffix of y obtained by removing x from y. Also, we denote by $x; seq$ the sequence obtained by extending x with the sequence of events seq. Saying that an extension y of x involves only processes from the set P means that all events in $(y - x)$ are only by processes in P. Runs x and y are *indistinguishable* for process p, if the subsequence of all events by p in x is the same as in y, the initial values of the local registers of p in x are the same as in y, and the (current) values of all the shared registers in x are the same as in y.

Process p *covers* a register in run x, if x can be extended by an event in which p writes to some register. If every process in a set of processes P covers a register in y then P is a set of *covering processes* in y. A *block write* by a set of covering processes P is an execution in which each process in P performs a single write (and nothing else). Notice that if process p covers register reg in run x then p covers reg in any extension of x which does not involve p. Also, if every process in P covers a different register then the order of writes does not matter since the resulting runs are indistinguishable to all the processes.

6.2 An impossibility result for deadlock-free mutual exclusion

A deadlock-free mutual exclusion algorithm for an unbounded number of processes using named registers was presented in [17]. We prove that there is no such algorithm using unnamed registers. These possibility and impossibility results imply Theorem 6.1.

Theorem 6.2. *There is no deadlock-free mutual exclusion algorithm using unnamed registers when the number of processes is not a priori known.*

Proof. We assume to the contrary that there is a deadlock-free mutual exclusion algorithm using unnamed registers assuming that the number of processes is not a priori known, and show that this assumption leads to a contradiction.

A process must write at least once before it enters its critical section. Otherwise, we can let the process enter its critical section, and since it left no trace, if we let other processes run, by deadlock-freedom, one of them will also enter its critical section violating the mutual exclusion requirement.

Let y be a run in which some process, say q, runs alone until it enters its critical section, and let $write(y, q)$ be the set of different registers to which q has written in y. Let P be a set of processes such that $q \notin P$ and $|P| = |write(y, q)|$. Since the number of processes is not a priori known (and thus can be as big as we need), we can always find such a set P, for any size of $write(y, q)$.

For each process $p \in P$, let $r.p$ denotes a run in which p runs alone until it first covers some register. Since all the registers are unnamed, we can let each process scan the registers in an order which ensures that: (1) for each process $p \in P$, in $r.p$ process p covers some register in $write(y, q)$, and (2) for each pair of processes $\{p, p'\} \subseteq P$, in $r.p$ and $r.p'$ processes p and p' cover different registers.

Since, for each $p \in P$ there are no writes in $r.p$, we can construct a run, called x, as follows: we pick an arbitrary process $p \in P$, and let it execute exactly the same steps as in $r.p$, then we pick another process from P and repeat this procedure. We continue until each process in P covers some register in $write(y, q)$. By construction, in x the processes in P together cover all the registers in $write(y, q)$. Let x' be an extension of x by a block write by P. By the deadlock-freedom property, there exists an extension z of x' in which only processes in P take steps and some process in P is in its critical section.

Next we construct a run, ρ, in which two processes are in their critical sections at the same time. We start the construction with the run x. Since no process writes a shared register in x, only processes in P participate in x and $q \notin P$, it follows that $x; y$ is a legal run. Let w be an extension of $x; y$ by a block write by P. Since the block write by P overwrites all the values written by process q in $x; y$, it follows that w and x' are indistinguishable for all the processes P. Thus, any extension of x' by processes in P is also a possible extension of w. In particular, $\rho = w; (z - x')$ is a legal run. By construction at the end of ρ two processes, $q \notin P$ and some process in P, are in their critical sections. A contradiction. □

6.3 An impossibility result and a space lower bound for obstruction-free consensus

In Section 4, we have presented an obstruction-free consensus for n processes using unnamed registers, where n is a (known) positive integer. We show that no such algorithm exists when the number of processes is not a priori known. Interestingly, it is shown in [25], how to construct an obstruction-free consensus algorithm using *named* registers when the number of processes is finite and not a priori known or even when the number of processes is unbounded. For a finite number of processes n, the question whether it is possible to solve obstruction-free consensus using $n - 1$ named registers is still open. We resolve this question for unnamed registers.

Theorem 6.3. *There is no obstruction-free consensus algorithm (1) when the number of processes is not a priori known using (an unlimited number of) unnamed registers, and (2) for $n \geq 2$ processes using $n - 1$ unnamed registers.*

Proof. We assume to the contrary that there is an obstruction-free consensus algorithm when the number of processes is not a priori known using unnamed registers (resp. for $n \geq 2$ processes using $n - 1$ unnamed registers), and show that this assumption leads to a contradiction.

A process that runs alone must write at least once before it decides. Otherwise, we let a process with input $v \in \{0, 1\}$ run alone until it decides v, Then (assuming the process never writes), we let a process with input $1 - v$ run alone until it decides on $1 - v$. This would violate the agreement requirement.

Let y be a run in which some process, say q, with input 0 runs alone until it decides 0, and let $write(y, q)$ be the set of different

registers to which q has written in y. Let P be a set of processes, all with input 1, such that $q \notin P$ and $|P| = |write(y, q)|$. Since the number of processes is not a priori known (resp. since there are n processes and only $n-1$ registers), we can always find such a set P, for any possible size of $write(y, q)$.

For each process $p \in P$, let $r.p$ denotes a run in which p runs alone until it first covers some register. Since all the registers are unnamed, we can let each process scan the registers in an order which ensures that: (1) for each process $p \in P$, in $r.p$ process p covers some register in $write(y, q)$, and (2) for each pair of processes $\{p, p'\} \subseteq P$, in $r.p$ and $r.p'$ processes p and p' cover different registers.

Since, for each process $p \in P$, there are no writes in $r.p$, we can construct a run, called x, as follows: we pick an arbitrary process $p \in P$, and let it execute exactly the same steps as in $r.p$, then we pick another process in P and repeat this procedure. We continue until each process in P covers some register in $write(y, q)$. By construction, in x the processes in P together cover all the registers in $write(y, q)$. Let x' be an extension of x by a block write by P. By the obstruction-freedom and the validity properties, there exists an extension z of x' in which (1) only some process $p \in P$ takes steps and (2) p decides 1.

Next we construct a run, ρ, in which two processes decide on different values. We start the construction with the run x. Since no process writes a shared register in x, only processes in P participate in x and $q \notin P$, it follows that $x; y$ is a legal run. Let w be an extension of $x; y$ by a block write by P. Since the block write by P overwrites all the values written by process q in $x; y$, it follows that w and x' are indistinguishable for all the processes P. Thus, any extension of x' by processes in P is also a possible extension of w. In particular, $\rho = w; (z - x')$ is a legal run. By construction at the end of ρ, q decides 0 and p decides 1. A contradiction. □

Corollary 6.4. *There is no obstruction-free implementation of a single named register using any number of unnamed registers.*

Proof. The corollary follows immediately from Theorem 6.3 and the fact that it is possible to construct an obstruction-free consensus algorithm using *named* registers when the number of processes is finite and not a priori known [25]. □

Remark: The k-set consensus problem is to design an algorithm for n processes, where each process starts with an input value from some domain, and must choose some participating process input as its output. All n processes together may choose no more than k distinct output values. The 1-set consensus problem, is the familiar consensus problem. It is possible to generalize Theorem 6.3, and prove that for every $k \geq 1$, there is no obstruction-free k-set consensus algorithm when the number of processes is not a priori known using (an unlimited number of) unnamed registers.

6.4 An impossibility result and a space lower bound for adaptive perfect renaming

In Section 5, we have presented an obstruction-free adaptive perfect renaming for n processes using unnamed registers, where n is a positive integer. Below we show that no such algorithm exists using only $n-1$ unnamed registers, or when the number of processes is not a priori known.

Theorem 6.5. *There is no obstruction-free adaptive perfect renaming algorithm (1) when the number of processes is not a priori known using (an unlimited number of) unnamed registers, and (2) for $n \geq 2$ processes using $n-1$ unnamed registers.*

Proof. We assume to the contrary that there is an obstruction-free adaptive perfect renaming algorithm when the number of processes is not a priori known using unnamed registers (resp. for $n \geq 2$ processes using $n-1$ unnamed registers), and show that this assumption leads to a contradiction.

A process that runs alone must write at least once before it decides. Otherwise, we let some process run alone until (by adaptivity) it acquires the new name 1. Then (assuming the process never writes), we let another process run alone until it also acquires the new name 1. This would violate the distinct new names requirement.

Let y be a run in which some process, say q, runs alone until it acquires the new name 1, and let $write(y, q)$ be the set of different registers to which q has written in y. Let P be a set of processes, such that $q \notin P$ and $|P| = |write(y, q)|$. Since the number of processes is not a priori known (resp. since there are n processes and only $n-1$ registers), we can always find such a set P, for any possible size of $write(y, q)$.

For each process $p \in P$, let $r.p$ denotes a run in which p runs alone until it first covers some register. Since all the registers are unnamed, we can let each process scan the registers in an order which ensures that: (1) for each process $p \in P$, in $r.p$ process p covers some register in $write(y, q)$, and (2) for each pair of processes $\{p, p'\} \subseteq P$, in $r.p$ and $r.p'$ processes p and p' cover different registers.

Since, for each process $p \in P$, there are no writes in $r.p$, we can construct a run, called x, as follows: we pick an arbitrary process $p \in P$, and let it execute exactly the same steps as in $r.p$, then we pick another process in P and repeat this procedure. We continue until each process in P covers some register in $write(y, q)$. By construction, in y the processes in P together cover all the registers in $write(y, q)$. Let x' be an extension of x by a block write by P. By the obstruction-free property and the adaptivity requirement, there exists an extension z of x' in which (1) only processes P take steps and (2) the process in P acquire the names 1 through $|P|$.

Next we construct a run, ρ, in which two processes acquire the name 1. We start the construction with the run x. Since no process writes a shared register in x, only processes in P participate in x and $q \notin P$, it follows that $x; y$ is a legal run. Let w be an extension of $x; y$ by a block write by P. Since the block write by P overwrites all the values written by process q in $x; y$, it follows that w and x' are indistinguishable for all the processes P. Thus, any extension of x' by processes in P is also a possible extension of w. In particular, $\rho = w; (z - x')$ is a legal run. By construction at the end of ρ, process $q \notin P$ and some other process in P both acquire the name 1. A contradiction. □

7 RELATED WORK

Our work is inspired by Michael O. Rabin's paper on solving the Choice Coordination Problem with k alternatives (k-CCP) [21]. In the k-CCP, n asynchronous processes must choose between k alternatives. The agreement on a single choice is complicated by the fact that there is *no a priori* agreement on names for the

alternatives. That is, each process has its own naming convention for the alternatives. Rabin has assumed that processes communicate by applying *read-modify-write* operations to k shared registers which do not have global names. The k different registers represent the k possible alternatives. For the case of $k = 2$ and $t = n-1$ where t is the possible number of faults, Rabin has presented a deterministic algorithm using $m = n+2$ symbols (for each register), proved a space lower bound of $m \geq (n/8)^{1/3}$ for deterministic algorithms, and contrasted these deterministic results with a randomized algorithm which, for m symbols, terminates correctly with probability $1 - 1/2^{m/2}$. Because of the use of read-modify-write operations, *none* of the algorithmic ideas from [13, 21] were found useful in our case.

The mutual exclusion problem was first introduced by Edsger W. Dijkstra in [10]. Dijkstra's original definition requires the algorithm to satisfy only the requirements of mutual exclusion and deadlock-freedom as defined in Section 3. In [7, 8], it is shown that any deadlock-free mutual exclusion algorithm for n processes using registers must use at least n shared registers. Dozens of interesting mutual exclusion algorithms and lower bounds are described in details in [22, 24]. The election problem (sometimes called the one shot mutual exclusion problem) is a special case of the mutual exclusion problem where only one process is allowed enter once its critical section. This process is the elected leader.

The consensus problem was formally defined in [20]. The impossibility result that there is no consensus algorithm that can tolerate even a single crash failure was first proved for the asynchronous message-passing model in [11], and later has been extended for the shared memory model with atomic registers, in [16]. Many related impossibility results can be found in [2]. Two extensively studied progress conditions are wait-freedom [14] and obstruction-freedom [15]. It is shown in [15] that obstruction-free consensus is solvable using registers. In [5], an obstruction-free consensus algorithm (which is not memory-anonymous) using single-writer registers and snapshot objects is presented. Recently, it has been shown in [27], that any obstruction-free consensus algorithm for n processes using registers must use at least $n - 1$ registers. The (one-shot) renaming problem was first solved for message-passing systems [1], and later for shared memory systems [4]. Several of the many papers on renaming are [3, 6, 9, 12, 18]. The impossibility result that there are no election algorithm and no perfect renaming algorithm that can tolerate a single crash failure was first proved for the asynchronous message-passing model in [1, 19], and later has been extended for the shared memory model in [26].

8 DISCUSSION

We have introduced several results for a model in which there is no lower-level agreement regarding the names of memory locations. The weak communication model considered, enables us to better understand the intrinsic limits for coordinating the actions of asynchronous processes.

We proved that unnamed registers (i.e, anonymous registers) are strictly weaker than named registers. Unnamed registers are the only non-trivial objects, that I know of, which are strictly weaker than the classical (named) atomic registers. It would be interesting to investigate the computational power of other unnamed objects, to consider models where, in addition to unnamed objects, a limited number of named objects are also available, and to explore models in which the number of objects and even their locations are not a priori agreed upon.

Several questions are left open: the existence of deadlock-free mutual exclusion algorithms for more than two processes, the existence of starvation-free mutual exclusion algorithms, finding tight space bounds for consensus and renaming, and finding solutions for additional problems under various progress conditions.

REFERENCES

[1] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *Journal of the Association for Computing Machinery*, 37(3):524–548, July 1990.

[2] H. Attiya and F. Ellen. *Impossibility Results for Distributed Computing*. Synthesis Lectures on Distributed Computing Theory. Morgan&Claypool, 2014. 162 pages.

[3] H. Attiya and A. Fouren. Algorithms adapting to point contention. *Journal of the ACM*, 50(4):144–468, 2003.

[4] A. Bar-Noy and D. Dolev. Shared memory versus message-passing in an asynchronous distributed environment. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 307–318, 1989.

[5] J. Bowman. Obstruction-free snapshot, obstruction-free consensus, and fetch-and-add modulo k. Technical Report Technical Report TR2011-681, Dartmouth College, 2011.

[6] A. Brodsky, F. Ellen, and P. Woelfel. Fully-adaptive algorithms for long-lived renaming. *Distributed Computing*, 24(2):119–134, 2011.

[7] J. E. Burns and A. N. Lynch. Mutual exclusion using indivisible reads and writes. In *18th annual allerton conference on communication, control and computing*, pages 833–842, October 1980.

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

[9] A. Castaneda, S. Rajsbaum, and M. Raynal. The renaming problem in shared memory systems: An introduction. *Computer Science Review*, 5(3):229–251, 2011.

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

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

[12] E. Gafni, M. Merritt, and G. Taubenfeld. The concurrency hierarchy, and algorithms for unbounded concurrency. In *Proc. 20th ACM Symp. on Principles of Distributed Computing*, pages 161–169, August 2001.

[13] D.S. Greenberg, G. Taubenfeld, and Da-Wei Wang Choice coordination with multiple alternatives. *In Proc. of the 6th International workshop on distributed algorithms*, LNCS 647, pages 54–68, 1992.

[14] M. P. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, January 1991.

[15] M. P. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. of the 23rd International Conference on Distributed Computing Systems*, page 522, 2003.

[16] M.C. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[17] M. Merritt and G. Taubenfeld. Computing with infinitely many processes. *Information and Computation*, 233:12 – 31, 2013.

[18] M. Moir and J. H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, October 1995.

[19] S. Moran and Y. Wolfstahl. Extended impossibility results for asynchronous complete networks. *Information Processing Letters*, 26(3):145–151, 1987.

[20] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

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

[22] M. Raynal. *Algorithms for mutual exclusion*. The MIT Press, 1986. Translation of: Algorithmique du parallélisme, 1984.

[23] E. Styer and G. L. Peterson. Tight bounds for shared memory symmetric mutual exclusion problems. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 177–191, August 1989.

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

[25] G. Taubenfeld. Obstruction-free consensus for unbounded concurrency. Unpublished manuscript, 2017.

[26] G. Taubenfeld and S. Moran. Possibility and impossibility results in a shared memory environment. *Acta Informatica*, 33(1):1–20, 1996.

[27] L. Zhu. A tight space bound for consensus. In *Proceedings of the Forty-eighth Annual ACM Symposium on Theory of Computing*, STOC '16, pages 345–350, 2016.