Linear Space Boostrap Communication Scheme

Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, Sergio Rajsbaum

Linear Space Bootstrap Communication Schemes

Carole Delporte-Gallet
U. Paris Diderot, France. cd@liafa.univ-paris-diderot.fr

Hugues Fauconnier
U. Paris Diderot, France. hf@liafa.univ-paris-diderot.fr

Eli Gafni
Computer Science Department, UCLA, USA. eli@ucla.edu

Sergio Rajsbaum
Instituto de Matemáticas, UNAM, Mexico. rajsbaum@math.unam.mx

Abstract

We consider a system of n processes with ids not a priori known, that are drawn from a large space, potentially unbounded. How can these n processes communicate to solve a task? We show that n a priori allocated Multi-Writer Multi-Reader (MWMR) registers are both needed and sufficient to solve any read-write wait-free solvable task. This contrasts with the existing possible solution borrowed from adaptive algorithms that require $\Theta(n^2)$ MWMR registers.

To obtain these results, the paper shows how the processes can *non-blocking* emulate a system of n Single-Writer Multi-Reader (SWMR) registers on top of n MWMR registers. It is impossible to do such an emulation with $n-1$ MWMR registers.

Furthermore, we want to solve a sequence of tasks (potentially infinite) that are sequentially dependent (processes need the previous task's outputs in order to proceed to the next task). A non-blocking emulation might starve a process forever. By doubling the space complexity, using $2n-1$ rather than just n registers, the computation is wait-free rather than non-blocking.

Keywords: shared memory, read/write registers, distributed algorithms, wait-free, space complexity, renaming.

1 Introduction

In many distributed algorithms it is assumed that the processes, $p_1, \ldots, p_n$, communicate using Single-Writer Multi-Reader (SWMR) registers, $R_1, \ldots, R_n$, so p_i knows it is the i-th process, and can write exclusively to R_i. However, often processes do not know their indexes, they know only their ids, and the number of possible ids N, is much bigger than the number of processes, n. In this situation, preallocating a register for each identifier would lead to a distributed algorithm with a very large space complexity. One would like the processes to run a *renaming* algorithm as a preprocessing stage, to obtain new ids from a smaller range, $M(n)$, that depends only on n, and then use these ids to index only $M(n)$ SWMR registers. Several wait-free renaming algorithms e.g. [6, 12] are known that reduce the name space to $M(n) = 2n-1$ (and this is the best than can be done, except for some exceptional values of n [14]). However, if the system did not allocate N SWMR register to start with, how do the processes communicate to run the renaming algorithm?

Such a question has been considered in the context of *adaptive* computation [7, 8, 25, 2, 4]. In these papers, n is unknown, and moreover processes "arrive" and "depart." When first solving adaptive renaming to allocate MWMR registers to processes, they obtain $\Theta(n^2)$ space complexity. The first basic question we address here is: In the most favorable situation, when n is known and we want to solve is a *task*, how much space do we need? Is solving renaming first unavoidable?

Recall that a task e.g. [20] is a one shot problem, where processes start with inputs and after communicating with each other, must decide on outputs that satisfy the task's specification. Famous examples of tasks are consensus and set agreement.

We show that indeed it is possible to solve any read-write wait-free solvable task with n MWMR registers. To obtain this result, the paper shows how the processes can *non-blocking* emulate a system of n SWMR registers on top of n MWMR registers. Moreover, as we explain, it is not hard to prove that no such emulation exists on less than n MWMR registers.

An application of the non-blocking emulation is that one can run any SWMR wait-free algorithm that solves a task on n MWMR registers. In particular, one can run directly a SWMR $(2n-1)$-renaming algorithm such as the one of [12] on top of n MWMR registers. This is a significant improvement over the previous adaptive $\Theta(n^2)$ space renaming algorithm, when that algorithm is instantiated to our context. Admittedly, the previous renaming algorithm will actually use fewer than n^2 registers but nevertheless since it is apriori unknown how the algorithm will evolve a preallocation of $\Theta(n^2)$ registers is necessary. Notice that as the simulation is non-blocking, a SWMR algorithm that solves a task on top of the n MWMR registers may incur some cost in time: a process may not be able to produce an output value, until another process finishes the algorithm and exits the emulation.

As said, with n registers we can solve $(2n-1)$-renaming, using these new names, with additional $2n-1$ registers each process can obtain a dedicated register, and from there on emulate a simulated write operation in a wait free-manner. Are $3n-1$ MWMR necessary to have a wait-free emulation of a write operation in a non-terminating environment? We show that a total of just $2n-1$ MWMR registers is sufficient, describing how the processes can *wait-free* emulate a system of n SWMR registers on top of $2n-1$ MWMR registers. We conjecture that $2n-1$ registers are the minimum possible for wait-free emulation. The wait-free emulation allows to solve a sequence of tasks (potentially infinite) that are sequentially dependent (processes need the previous task's outputs in order to proceed to the next task). A non-blocking emulation might starve a process forever. By doubling the space complexity, using $2n-1$ rather than just n registers, the computation is wait-free rather than non-blocking.

The paper also describes an algorithm to broadcast the value of one of the processes, using $n/2$ MWMR registers. The algorithm, which seems interesting in itself, illustrates the way information propagates in our emulations. It is simple, but the proof is subtle.

At the end of the paper we briefly discuss why we believe our techniques will be useful for dynamic systems of bounded concurrency n e.g. [4]. In such a system, any number of processes arrive compute and depart, yet, at any point in time the number of arrivals exceeds the number of departures by at most n.

We stress that our interest is in space complexity, our emulations are not particularly efficient in terms of step complexity. Also, there are various previous papers dealing with a similar problem, but in the adaptive setting. In [10] there is a definition similar to our emulation problem, as "store-collect" of a (key,value) problem. Another definition of store-collect, by equivalence to an array (hence more like our problem specification) appears in [3]. Similarly, solutions without explicitly solving renaming first appear in [9], although not very space efficient.

There is a long history of space complexity results, starting with the mutual exclusion lower bound of [13] and even before there was interest in this (see references herein). In this context, [26] considers renaming with the same motivation that we do, and shows that $2\lceil logn\rceil + 1$ registers are sufficient to solve it, and a corresponding lower bound within a constant factor, but their model is easier than ours. There are various algorithms and lower bounds on the number of registers needed to solve specific problems e.g.

[15, 16, 18, 25], but we are not aware of general emulations. In [23] there is a lower bound for non-blocking implementations, but it is $n-1$ registers. In special cases it can be beaten [5]. In [17] a $\Omega(\sqrt{n})$ space complexity lower bound for a randomized nonblocking implementation of consensus is presented.

2 Model

We assume a standard asynchronous shared-memory model of computation with n processes communicating by reading and writing to a fixed set of shared registers [11, 21]. The processes have unique ids in $\{1, \ldots, N\}$, with $N >> n$. Processes may take a finite or an infinite number of steps but we assume that at least one process takes an infinite number of steps.

The shared memory consists of a set of atomic Multi-Writer Multi-Readers (MWMR) registers. We assume that processes can read and write any MWMR register and these operations are atomic [19]. For short, we usually omit the term atomic. If r is such a MWMR register, a process can write x on r using $write(r, x)$, and read on r using $read(r)$. A process executes its code taking three types of atomic steps: the read of a register, the write of a register, and the modification of its local state.

We consider also more powerful operations to read the registers. A *collect* is an iterative read of all registers. A *scan* returns a snapshot, an instantaneous view of the registers. In [1], there are non blocking and wait free linearizable implementations of the scan. A non-blocking implementation of scan ($NBScan$) can be obtained repeating $Collect$ operations until two of them return the same values. A scan wait-free implementation ($Scan$) is more involved, and has to embed a snapshot with the write. So in this case we call the *write*, *update* ($update(r, v)$ updates the register r with the value v).

Two progress conditions that have received much attention are non-blocking and wait-free. The *non-blocking* progress condition states that when there are concurrent operations at least one process terminates its operations. The wait-free [19] progress condition states that each process terminates its operations in a bounded number of its own steps.

We consider also a model in which each process has its own atomic Single-Writer Multi-Reader register (SWMR). Process p can write a value x in its SWMR register with the operation $\textsc{write}(p, x)$ and all the processes may read the value in the SWMR register of p with $\textsc{read}(p)$.

We say that we have a non-blocking (resp. wait-free) *emulation* of the n process SWMR model using m MWMR registers if we implement WRITE and READ in terms of *write* and *read* such that the implementation is linearizable [22] and the progress condition is non-blocking (resp. wait free).

A regular SWMR register [24], is a weaker type of register. A SWMR register is *regular* if when no write is concurrent with a read operation, the read returns the current value in the register. Otherwise the read returns any value written by a concurrent write operation or the last value in the register before the concurrent writes. With regular SWMR registers it is possible to wait-free free implement atomic SWMR register [21, 24]. So to emulate non-blocking (resp. wait-free) the n process SWMR register model using m MWMR registers it is enough to emulate non-blocking (resp. wait-free) n process model with regular SWMR register model.

3 Preliminaries

We consider algorithms that have the same structure as algorithm in Figure 1. Processes share a set $\mathcal{R}$ of MWMR registers. Each process maintains a variable $View$. Each process repeatedly reads all registers (function $Collect()$) and updates its variable $View$ by adding all it has just read in $View$ together with some other values in variable *input*, and then writes it in some registers.

For each $R \in \mathcal{R}$, R^τ denotes the value of register R at time τ. Similarly, $View_p^\tau$ denotes the value of variable $View$ of p at time τ.

We say that v is *eventually forever* in R if there is a time τ such that for all time $\tau' > \tau$, $v \in R^{\tau'}$.

```
function $Collect()$:
1  $V = \emptyset$
2  for all $R$ in $\mathcal{R}$
3      $V = V \cup read(R)$
4  return $V$

main loop:
   repeat forever
5      $View = Collect() \cup View \cup input$
6      Let (deterministically) $Reg$ be some register in $\mathcal{R}$
7      $write(Reg, View)$
```

Figure 1: Generic algorithm.

We have directly from the algorithm:

Observation 1 *For all processes p and for all times τ and τ', if $\tau < \tau'$ then $View_p^{\tau} \subseteq View_p^{\tau'}$*

Following [13], we say that register R is *covered* by process p at some time τ if the next register that p writes after time τ is R: more precisely p *covers* R at time τ if, at time τ, the next writing of p (Line 7) is on register R and p has begun the *Collect* in Line 5. Note that if p does not cover R, before writing R, p reads the value of R (by *Collect* in Line 5) and then it will write in R at least this value.

If some register is covered the value of this register may be lost. So, by extension, we say that R is *V–covered* at time τ if all processes covering R at time τ are going to write sets containing V in register R. If R is V-covered at time τ, next writings contains V (by processes covering R) or contains R^{τ} (by processes not covering R that read R before writing). We have:

Lemma 2 *If, at some time τ, register R is V–covered at time τ then for all $\tau' \geq \tau$, $(R^{\tau} \cap V) \subseteq R^{\tau'}$.*

Proof. By induction on the number k of writings of R made after time τ, we prove that R contains $R^{\tau} \cap V$. For $k = 0$ it is clear. Consider any writing of R after time τ and assume this writing is made, say, by process q. Consider the two cases:

- either q does not cover R at time τ and then q makes a *Collect* before writing R. This *Collect* occurs after time τ and by induction hypothesis it returns a value containing $R^{\tau} \cap V$. Then process q writes a superset of $R^{\tau} \cap V$ in register R.
- or q covers R at time τ and $V \subseteq View_p$ then q writes a superset of $R^{\tau} \cap V$ in R.

□

If no process covers R at time τ, then by definition R is V-covered for any set V, then by the previous Lemma after time τ, R contains forever R^{τ}. In particular:

Lemma 3 *If R is not covered at time τ and $v \in R^{\tau}$ then v will be eventually forever in R.*

Let $\mathcal{R}_\infty$ be the set of registers infinitely often written:

Lemma 4 *If v is eventually forever in R then v is eventually forever in all registers in $\mathcal{R}_\infty$.*

Proof. There is a time τ after which v is forever in R, after this time all *Collect* contains v and then all next writing will contain v. Hence all registers infinity often written will contain v. □

Lemma 5 *If at time τ, $card\{R \in \mathcal{R} | v \in R^{\tau}\} \geq n$, then v will be forever in all registers in $\mathcal{R}_\infty$*

Proof. Let X be the set of at least n registers containing v and p be the last process that wrote before time τ in any register in X. Just after this writing p does not cover any register. As we have n processes, by the pigeon hole principle at least one register in X is not cover. Then by Lemma 3, v will be forever in some register and by Lemma 4, v will be forever in all $\mathcal{R}_\infty$. □

4 Broadcast

We give here a very weak definition of broadcast. Essentially this definition ensures that the value of some process will be known by all processes making an infinity of steps. More precisely, we assume that each process p has a value v_p to broadcast, the broadcast is defined by way of a primitive *deliver* operation returning a set of values. The broadcast ensures:

- *(integrity)* if v belongs to the set of values returned by some *deliver* then $v = v_p$ for some process p,
- *(convergence)* there is a value v and a time τ after which every set returned by any *deliver* contains v.

```
CODE FOR PROCESS $p$
Shared variable :
        array of $m$ MWMR register : R
Local variable:
        set of Values $View = \{v_p\}$
        set of Values $Deliver = \emptyset$
function $Collect()$:
    variable: set of Values $V = \emptyset$
        for $i$ from 1 to $m$ do
            $V = V \cup read(R[i])$
        return $V$
Task main:
    repeat forever
        for $i = 1$ to $m$
            $View = View \cup NBScan()$;
            $write(R[i], View)$
Task Deliver:
        forever $Deliver = Collect()$
```

Figure 2: Broadcast with m MWMR registers

Here processes share m MRMW registers $R[i]$, $1 \leq i \leq m$ and write successively each register.

The algorithm of Figure 2 is a simple application of the generic algorithm of Figure 1. But here in, the main loop, we use a non-blocking scan *NBScan* instead of *Collect*. $NBScan()$ returns V, a snapshot of the registers, such that there is a time τ such that $R[i]^{\tau} = V[i]$. $NBScan()$ is only assumed to be non-blocking. Implementation of such non-blocking snapshot is easy [1]. As *NBScan* is a particular form of *Collect* all previous Lemmata from Section 3 apply. For a snapshot V, $V[i]$ is the value returned for register $R[i]$ and abusing notation, V may denote $\cup_{1 \leq i \leq m} V[i]$.

A *Deliver* reads all the registers and returns the union of the values read in the registers. Note, using $Collect()$ that *Deliver* always terminate. We prove the main property:

Theorem 6 *Algorithm of Figure 2 implements broadcast if* $m > \frac{n}{2}$

The proof is rather subtle, it is in Appendix A.

5 Non-Blocking emulation of SWMR registers

We first describe the emulation, and then the lower bound, in Section 5.2.

5.1 The algorithm

The algorithm in Figure 3 is a non-blocking emulation of regular SWMR registers for n processes using an array R of m MWMR registers, if $m \geq n$.

In the following to distinguish between the writings of MWMR registers and the emulation of SWMR writings, we denote the first ones using lower case and the second ones with upper case. To make its k-WRITE, a process p adds the value to WRITE (in fact the value, its identity and its timestamp k) to its variable $View$. The WRITE ends when the value to be WRITTEN is in all the m registers. A READ of value WRITTEN by process q collects the values present in registers and returns the value from q with a maximal timestamp among all the values from q.

As in the generic algorithm of Figure 1, each process p maintains a variable $View$ containing all the information it knows. Iteratively, each process reads all the m registers by a non-blocking scan and accordingly updates its variable $View$ before writing it in the next register in cyclic order. The WRITE of v terminates as soon as v is in n registers. Here instead of *Collect*, we use *NBScan*, a non blocking *Scan*, as described in Section 2.

```
Shared variable :
        array of $m$ MWMR-register : R
// To ensure non-blocking we assume $m \geq n$
Local variable:
        set of Values $View = \emptyset$
CODE FOR PROCESS $p$

init k=0;
//$NBScan$ returns a snapshot of the shared memory
WRITE$(p, x)$:
1   $v = (x, p, k)$
2   $next = 0$
3   $View = View \cup \{v\}$
4   do
5       $Snap = NBScan()$
6       $View = Snap \cup View$
7       $write(R[next], View)$
8       $next = (next + 1) \bmod m$
9   until $(card\{r|v \text{ in } Snap[r]\} \geq n)$
10  $k = k + 1$

READ$(q)$:
11  $View = Collect()$
12  return $x$ such that $(x, q, u) \in View$ with maximal $u$
```

Figure 3: Non blocking implementation of regular SWMR registers for n processes.

The main point here is that as we have n processes and at least n registers then at least one register will not be covered and then all the values contained in this register will eventually be present in all registers and will remain forever in all registers. Hence as soon as a value is present in all registers the WRITE is terminated because afterwards the *Collect* of every READ will contain this value.

First we prove the safety properties of the implementation of regular register.

We say that the WRITE of v *succeeds* at time τ if there is some register such that after time τ, v is forever in this register. By extension, we say that the WRITE of v succeeds if there is a time at which the WRITE of v succeeds or equivalently if v is eventually forever in R. Directly from the algorithm we get the following lemmas:

Lemma 7 *Let $v = (x, p, k)$ and $v' = (y, p, k')$ such that $k \geq k'$, if v succeeds at time τ, then v' succeeds at time τ too.*

By Lemma 4 and the code of the algorithm:

Lemma 8 *If the* WRITE *of v succeeds at time τ, after this time v is returned by every $Collect$.*

Lemma 9 *Let S be the set of all values (x, p, k) that succeed at time τ and K the maximal k over all (x, p, k) in S, then* READ *of p returns the value $v = (x, p, k) \in S$ with $k \geq K$ maximal.*

From Lemma 5:

Lemma 10 *If at some time τ, v is in n registers, then the* WRITE *of v succeeds at time τ.*

Lemma 11 (safety) *Any* READ(p) *returns the last value x such that* WRITE(p, x) *terminates before the beginning of the* READ*, or a value x such that* WRITE(p, x) *is concurrent with the* READ*.*

PROOF. Assume x is the kth write of p, let v be (x, p, k). Consider any READ(p) and let E be the set of values returned by the *Collect* made for this READ. By Lemma 8 all values for which the WRITE has succeeded are in E. WRITE(p, x) returns when, for a $NBScan$ (Line 5), v belongs to n registers (Line 9) at some time. Then by Lemma 10, v succeeds by the time of the $NBScan$ and E contains all values for which the WRITE has terminated. Lemma 9 proves that the value returned by the READ(p) is either the last value for which the WRITE by p has terminated or a value for which the WRITE is concurrent. □

Now we prove that the algorithm is non-blocking. First as *Collect* is wait-free, any READ is wait-free too:

Lemma 12 *Any* READ *made by a process that takes an infinite number of steps terminates.*

By Lemma 4, and the fact that the WRITE ends when the value is in all registers:

Lemma 13 *Assume the registers are written infinitely often, if the* WRITE *of v succeeds then v will eventually be forever in all registers and if the process that* WRITES *v takes an infinite number of steps, the* WRITE *terminates.*

Lemma 14 *If the registers are written infinitely often then an infinity of* WRITE *terminate.*

PROOF. By contradiction assume the contrary: there is a time τ after which no WRITE terminates. When a process has terminated all its WRITEs it stops writing the registers, then there is at least one process that takes an infinite number of steps and does not terminate the WRITE of some v. By Lemma 13, there is time at which all registers will contain values for which the WRITE is not terminated. By pigeon hole principle one of these registers is not covered and contains a value for which the WRITE is not terminated, by Lemma 3 the WRITE of this value succeeds, and by Lemma 13, the WRITE of this value terminates –a contradiction. □

If after some time there is no writing of the registers, being non-blocking any $NBScan$ returns:

Lemma 15 *If some process that takes an infinite number of steps is stuck forever on a $NBScan$ then the registers are written infinitely often.*

Lemma 16 (non-blocking) *If $m \geq n$, and the* WRITE *of v by a process p that takes an infinite number of steps does not terminate, then infinitely often some* WRITE*s terminate.*

PROOF. By contradiction, assume that the WRITE of some process p that takes an infinite number of steps does not terminate and only a finite number of WRITEs occurs. Then by Lemma 14, there is a time after which no registers are written and by Lemma 15, p may not be stuck on a $NBScan$ and hence p makes progress in its code and hence writes registers infinitely often. By Lemma 14 an infinity of WRITE terminates —a contradiction □

Lemmas 11 and 16 prove that the algorithm in Figure 3 is a non blocking emulation of regular SWMR registers for n processes from n MWMRregisters. We now use the classical wait free transformation [21, 24] from regular to atomic registers to conclude:

Theorem 17 *There is a non blocking emulation of SWMR registers for n processes from n MWMR registers.*

5.2 Lower bound

We prove in this section that we cannot emulate SWMR registers for n processes with less than n MWMR registers.

Lemma 18 *SWMR registers for n processes cannot be emulated with $n-1$ MWMR registers.*

PROOF. Consider a set of n processes $p_1, \ldots, p_n$. By contradiction, assume we can emulate the SWMR system with $(n-1)$-MWMR atomic registers. We construct inductively a run e where this assumption is not satisfied. For this we construct by induction on k a partial run e_k and a set $\mathcal{R}_k$ of k registers each being covered by processes $p_1, \ldots, p_k$.

$(k = 1)$: In e_1 only process p_1 takes steps and its code is WRITE(p_1, p_1).

Claim: *p_1 has to write in some register.*

PROOF. By contradiction assume p_1 does not write in any register and assume the code of p_n is READ(p_1). Once p_1 ends its WRITE, p_n takes steps and does not find any value written by p_1, contradicting the semantics of a register. □

In e_1, only p_1 takes steps and stops just before its first writing of a register, say R_1. Then p_1 covers R_1 and $\mathcal{R}_1 = \{R\}$.

$(k < n-1)$: By induction let e_k be such that $\{p_1, \cdots, p_k\}$ covers each register in the set of k registers $\mathcal{R}_k$. Run e_{k+1} extends partial run e_k for the process p_{k+1} that executes the code WRITE(p_{k+1}, p_{k+1}).

Claim: *p_{k+1} has to write in some register not in $\mathcal{R}_k$.*

PROOF. By contradiction assume p_{k+1} does not write any register not in $\mathcal{R}_k$ and assume the code of p_n is READ(p_{k+1}). p_{k+1} ends its WRITE and has only written registers in $\mathcal{R}_k$, then each process in $\{p_1, \ldots, p_k\}$ executes one step and overrides each register in $\mathcal{R}_k$. Then p_n executes the code for READ(p_{k+1}). But this execution is indistinguishable (for all processes different from

p_{k+1}) from one in which p_{k+1} does not make any WRITE and the READ may not return the value WRITTEN by p_{k+1}. □

In e_{k+1}, p_{k+1} takes steps and stops before writing a register that is not in $\mathcal{R}_k$, say R_{k+1}. Then p_{k+1} covers R_{k+1}. Define $\mathcal{R}_{k+1} = \mathcal{R}_k \cup R_{k+1}$.

When $k = n-1$, $\mathcal{R}_{n-1}$ contains all the MWMR registers. Now, consider process p_n and assume it runs the code WRITE(p_n, p_n). Run e_n is an extension of $e_n - 1$ in which p_n takes steps until it ends its WRITE and takes no other steps. Each process executes one step and overrides each register in $\mathcal{R}_{n-1}$. At this point the value WRITTEN by p_n is not in the local memory of any process (except p_n) and in particular it is not in the local memory of p_1. Then p_1 ends its WRITE(p_1, p_1), at the end of this WRITE, the value WRITTEN by p_n is not in any MWMR registers and the run is indistinguishable (for all processes different from p_n) from the same run in which p_n does not take any WRITE. Now if p_1 runs READ(p_n), p_1 cannot get the value written by p_n. □

6 Wait-free emulation of SWMR registers

The previous emulation is only non-blocking. Using $3n-1$ MWMR registers with help of the simulation of n non-blocking SWMR registers it is easy to simulate n *wait-free* SWMR registers. For this, the $3n-1$ registers are partitioned into a set W of n registers and a set PR of $2n-1$ registers. The n MWMR registers of W are used to (non-blocking) simulate n SWMR register with the algorithm of Figure 3. With these n SWMR registers we can run a renaming algorithm [6, 12]. In fact as such an algorithm always terminates in a finite number of steps it is easy to verify that the non-blocking simulation is enough to ensure that the renaming terminates. Hence each process p of the n processes gets an unique identity $id(p)$ in the set $\{1, \cdots, 2n-1\}$. To WRITE a value v, p will write in the $id(p)$th register of PR. As p is the unique writer the simulation is wait-free. But it is possible to reduce the number of MWMR registers to $2n-1$. In the algorithm we propose we eventually get an unique identity in the set $\{1, \ldots, n\}$.

```
Shared variable :
    array of $n-1$ MWMR-register : W
    array of $n+1$ MWMR-register : PR
Local variable:
    set of Values $View = \emptyset$
//$View.proc$: set of processes in $View$ ordered by names
//$Update$, $Scan$: wait-free snapshot
CODE FOR PROCESS $p$

init k=0;

WRITE$(p, x)$:
  $v = (x, p, k)$
  $next = 0$
  $nextReg = W[next]$
  $View = View \cup \{v\}$
  $Name$ = index of $p$ in $View.proc$
  do
      $Snap = Scan()$      /*of $W$ and $PR$*/
      $View = Snap \cup View$
      if  $(Snap[card(View.proc)] = \perp)$ then
          $Name$ = index of $p$ in $View.proc$
          $Update(PR[card(View.proc)], View)$
      else
          if $next = n$ then $update(PR[Name], View)$
          else $update(W[next], View)$
          $next = (next + 1)$ mod $(n+1)$
  until $(card\{r | v$ in $Snap[r]\} \geq n)$
  $k = k + 1$

READ$(q)$:
  $View = Collect()$
  return $x$ such that $(x, q, u) \in View$ with maximal $u$
```

Figure 4: Wait-free emulation with $2n$ MWMR registers for n processes.

We present first an algorithm with $2n$ MWMR registers. The algorithm of Figure 4 has a structure similar to the previous algorithms. Each process maintains a variable $View$ containing all information it knows. Here the processes access shared variables using wait free snapshot by means of primitives *Update* and *Scan*. Implementations of such primitives are described for example in [1].

For a set V of values *V.proc* denotes the list of all processes occurring in V (process q occurs in V if there is any $v = (x, q, s)$ in V). For convenience all the lists of processes are ordered by process identities. The *index* of process p in an ordered list of processes is the rank of p in the list. Indexes begins on 0, for example, for $1, 3, 8, 15$ the index of 1 is 0 and index of 3 is 1.

In the algorithm the processes share an array W (Working registers) of $n - 1$ MWMR-registers ($W[0], \ldots, W[n-2]$) and an array PR (Personal Registers) of $n + 1$ MWMR-registers ($PR[0], \ldots, PR[n]$).

Personal registers play a double part: they give the supposed number of participants and give a personal register for each participant. For this, the last cell different from $\perp$ gives the supposed number and list of participants: if $PR[a]$ is this last cell then the supposed number of participants is a and the list of participants is $PR[a]$*.proc*. In the ordered list of assumed participants process p determines its index (variable $Name$) and will consider $PR[Name]$ as its personal register (Lines 5 and 10).

As usual to WRITE a value x (assuming it is the kth), a process adds $v = (x, p, k)$ to its $View$ and successively writes (by *Update* and after updating its $View$ by *Scan* in Line 8) to the $n - 1$ registers W (Line 14) and in its personal register $PR[Name]$(Line 13). But it has to check that its $Name$ is correct. After each *Scan* a process will verify that the number of participants corresponds to the contents of PR registers (Line 9): if the number of participants is a then $PR[a]$ must be the last cell different from $\perp$. If it is the case, its $Name$ is up to date and it writes the next register (Line 13 and 14). If it not the case, the index of the last non $\perp$ cell in PR is less than the number of participants that p has seen, then p writes in the correct cell of PR ($PR[a]$ if a is the number of participants seen by p) (Line 11) and determine its new index $Name$ in its list (Line 10).

As before the WRITE terminates when v is in at least n registers (condition in Line 16): when v is in n registers at least one of these registers is not covered. Roughly speaking, WRITE is wait-free because eventually the last cell distinct of $\perp$ will correspond to the actual number of participants, and the list of participants in this cell will be the actual list of all participants, hence processes will have a personal register in which it will be alone to write.

The same arguments proving the safety properties of the non-blocking algorithm of Figure 3 apply here (essentially the WRITE of v terminates when v is in at least n registers by a non covering argument then v will be returned by all *Scan*). Then we restrict ourselves to prove the wait-freedom.

For each register r, *r.proc* is the list of all processes occurring in r.

Lemma 19 *There is a time after which each register r of $PR[0], ..., PR[n-1]$ is written by at most one process.*

PROOF. Let *max* be the greatest i such that $PR[i] \neq \perp$. By definition of the index, *max* may not be the $Name$ of any register, then $PR[max]$ may only be written by processes in Line 11 for which for the previous *Scan* $PR[max]$ was $\perp$. Hence this register is written only a finite number of time. Consider time τ after which only processes that make an infinite number of *Scan*/*Update* take steps and after which $PR[max]$ is no more written.

Claim There are a set *Participant* of size *max* and a time $\sigma > \tau$ after which: (1) $PR[max].proc = Participant$ and (2) for every process p that makes an infinite number of *Scan*/*Update*, $View_p.proc = Participant$.

PROOF. Let *Participant* be the value of $PR[max].proc$ a time τ.

Let p be a process that makes an infinite number of *Scan*/*Update*, p will make some *Scan* after time τ getting $PR[max]$. Then there is a time τ_p after which $PR[max].proc \subseteq View_p.proc$. Then $PR[max].proc \neq View_p.proc$ is equivalent to $max \neq card(View_p.proc)$. If $Participant = PR[max].proc \neq View_p.proc$ then $max < card(View_p.proc)$ and p will write in $PR[card(View_p.proc)]$ contradicting the definition of $PR[max]$. Let time σ be max of τ_p for all p making an infinite number of *Scan*/*Update*, after time σ conditions (1) and (2) of the claim are satisfied. □

Let i such that $0 \leq i \leq n$ and assume that $PR[i]$ is written after time σ, then by definition of max, we can assume that $0 \leq i < max$. If, after time $\sigma > \tau$, some q writes $PR[i]$, by definition of τ, this process makes an infinite number of $Scan/Update$. By the claim, $View_q.proc = Participant$, and if q writes in $PR[i]$ that means that the index of q in $Participant$ is i and as each process making an infinity of $Scan/Update$ has an unique index in $Participant$, q is the only writer of $PR[i]$. □

Lemma 20 *The algorithm in Figure 4 emulates wait-free n SWMR registers with $2n$ MWMR registers for n processes.*

Proof. We prove only the wait-freedom. Assume a process p that takes an infinite number of steps tries to WRITE $v = (x, p, k)$ in its register and does not succeed. Then (v, p, k) in inserted in $View_p$ and by Lemma 1, (v, p, k) remains forever in $View_p$

By the algorithm, p executes an infinite number of $Scan/Update$ and writes $View_p$ an infinite number of times in at least one register of PR. By Lemma 19, there is a register $PR[i]$ and a time τ after which p writes infinitely often in $PR[i]$ and p is the only writer of $PR[i]$. Then v will be forever in $PR[i]$. After time τ, every process q that executes $Scan/Update$ reads $PR[i]$ and includes it in its $View_q$. Then all processes that write after time τ in some register will write v in this register too. As we assume that the WRITE of p does not terminate, at least p writes in each register of W. Therefore, there is a time after which v will be forever in all the $n-1$ registers W and in register $PR[i]$, then for any $Scan$ v will be in at least n registers and the WRITE of v ends. □

Notice that the information in $PR[n]$ can be easily integrated to $PR[n-1]$. So we can use only $2n-1$ MWMR registers. The shared MWMR registers array W is of size $n-1$ and array PR is of size n.

Finally, we replace Line 9 to Line 11 from algorithm in Figure 4 with Lines in Figure 5 :

Theorem 21 *The algorithm in Figure 5 emulates wait-free n SWMR registers with $2n-1$ MWMR-registers for n processes.*

```
9.1  if $Name \neq$ index of $p$ in $View.proc$ then
9.2      $Name =$ index of $p$ in $View.proc$
9.3  if $(card(View.proc) = n$
                 and $card(PR[n-1].proc) = n-1)$
         or $(card(View.proc) < n$
                 and $Snap[card(View.proc)] = \bot))$
9.4      then
9.5      if $(card(View.proc) = n)$ then
9.6          $update(PR[n-1], View)$
9.7          else
11           $update(PR[card(View.proc)], View)$
```

Figure 5: Wait-free emulation with $2n-1$ MWMR registers for n processes.

7 Concluding remarks

We have seen how n processes emulate SWMR non-blocking using n-registers and wait-free using $2n-1$ MWMR registers. What if we have M processes $M >> n$ (M possibly infinite) but we have the notion of arrivals and departures of processes? Processes have to solve the task only under the condition that the number of processes that invoked the task (arrived) minus the number of processes that obtain an output (departed) is at any point of time less or equal to n. Suppose that we know that a task T is read-write non-blocking (resp. wait-free) solvable in the SWMR model under the assumption of n-concurrency. Can we solve T with just n (resp. $2n-1$) MWMR registers, without indefinite postponement, i.e. the step complexity of a process until it gets an output will be, like in the SWMR model, a function of n rather than M?

We observe that the non-blocking simulation is in fact n-concurrent. Thus, for the non-blocking case, the answer is positive. Concerning the wait-free simulation it is more tricky, but we conjecture that the answer is positive too.

References

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, 1993.

[2] Yehuda Afek and Yaron De Levie. Efficient adaptive collect algorithms. *Distributed Computing*, 20:221–238, 2007.

[3] Yehuda Afek, Gideon Stupp, and Dan Touitou. Long-lived adaptive collect with applications. In *Proceedings of FOCS*, FOCS '99, pages 262–272. IEEE, 1999.

[4] Marcos K. Aguilera. A pleasant stroll through the land of infinitely many creatures. *SIGACT News*, 35(2):36–59, June 2004.

[5] James Aspnes, Hagit Attiya, and Keren Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM*, 59(1):2:1–2:24, March 2012.

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

[7] Hagit Attiya and Arie Fouren. Polynominal and adaptive long-lived (2k-1)-renaming. In *Proceedings of the 14th International Conference on Distributed Computing*, DISC '00, pages 149–163, London, UK, UK, 2000. Springer-Verlag.

[8] Hagit Attiya and Arie Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31(2):642–664, February 2002.

[9] Hagit Attiya and Arie Fouren. Algorithms adapting to point contention. *J. ACM*, 50(4):444–468, July 2003.

[10] Hagit Attiya, Arie Fouren, and Eli Gafni. An adaptive collect algorithm with applications. *Distributed Computing*, 15(2):87–96, July 2002.

[11] Hagit Attiya and Jennifer Welch. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics.* John Wiley & Sons, 2004.

[12] Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *PODC*, pages 41–51. ACM Press, 1993.

[13] James E. Burns and Nancy A. Lynch. Bounds on shared memory for mutual exclusion. *Inf. Comput.*, 107(2):171–184, 1993.

[14] Armando Castañeda and Sergio Rajsbaum. New combinatorial topology bounds for renaming: the lower bound. *Distributed Computing*, 22(5-6):287–301, 2010.

[15] Panagiota Fatourou, Faith Fich, and Eric Ruppert. Space-optimal multi-writer snapshot objects are slow. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 13–20, New York, NY, USA, 2002. ACM.

[16] Panagiota Fatourou, Faith Ellen Fich, and Eric Ruppert. Time-space tradeoffs for implementations of snapshots. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, STOC '06, pages 169–178, New York, NY, USA, 2006. ACM.

[17] Faith Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5):843–862, September 1998.

[18] Maryam Helmi, Lisa Higham, Eduardo Pacheco, and Philipp Woelfel. The space complexity of long-lived and one-shot timestamp implementations. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '11, pages 139–148, New York, NY, USA, 2011. ACM.

[19] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, January 1991.

[20] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(2):858–923, 1999.

[21] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2008.

[22] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[23] Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for nonblocking implementations. *SIAM J. Comput.*, 30(2):438–456, April 2000.

[24] Leslie Lamport. On interprocess communication; part I and II. *Distributed Computing*, 1(2):77–101, 1986.

[25] Mark Moir. Fast, long-lived renaming improved and simplified. *Sci. Comput. Program.*, 30(3):287–308, March 1998.

[26] E. Styer and G. L. Peterson. Tight bounds for shared memory symmetric mutual exclusion problems. In *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, PODC '89, pages 177–191, New York, NY, USA, 1989. ACM.

A Broadcast: Proof of Theorem 6

If a process takes an infinity number of steps then its *Deliver*s terminate, and the Integrity property is trivial. We now prove the convergence.

Recall that we assume that there is at least one process that takes an infinite number of steps.

Lemma 22 *At least one process makes an infinite number of $NBScan$ and all registers are written infinitely often.*

Henceforth $\mathcal{R}_\infty$ is here the set of all registers.

If all registers contain some set of values V, if the number of processes covering registers with values W such that $V \not\subseteq W$ is less than m, the number of registers, then V will be forever contained in all registers.

Lemma 23 *If at some time τ, let $V \subseteq \bigcap_{1 \leq j \leq m} R[j]^\tau$ and let $Q = \{q \in \Pi | V \not\subseteq View_q^\tau\}$, if $card(Q) < m$ then there is a time $\tau'' > \tau$ such that for all $\tau' > \tau''$, $V \subseteq \bigcap_{1 \leq j \leq m} R[j]^{\tau'}$.*

Moreover there is a time after which for all processes p that take an infinite number of steps $V \subseteq View_p$.

Proof. If $card(Q) < m$ then by pigeon hole principle at least one register, say $R[j]$, is $V-$*covered* then by Lemma 2, $V = V \cap R[j]^\tau$ will be forever in this register. By Lemma 4, eventually V will be in all registers. Then if p takes an infinite number of steps it eventually makes a $NBScan$ that returns a superset of V and by Lemma 1 there is a time after which V is forever in $View_p$. □

Now consider some properties on sets written in registers. From Lemma 1 and the fact that $View$ is a subset of the set of initial values and this set is finite we get:

Lemma 24 *For all processes p, there is a set V of values such that there is a time after which forever $View_p = V$.*

A set of values E is said to be *stable* if E is written infinitely often in some register. By Lemma 22 and the fact that there is a finite set of values, there is at least one stable set.

Then, directly from the definition:

Lemma 25 *E is stable if and only if there is a process p that takes an infinite number of steps and a time τ after which we have forever $E = View_p$.*

As a process writes always some values, a stable set is not the empty set.

Lemma 26 *If E is stable then $E \neq \emptyset$.*

Hence, let τ_0 the time after which only stable sets are written in registers.

Stable sets may be ordered by inclusion, for this order a stable set E is minimal if and only if for all stable sets E' if $E' \subseteq E$ then $E' = E$. As the set of stable sets being finite, minimal stable sets always exist.

Consider a minimal stable set M, and assume a time $\tau \geq \tau_0$. Let $\mathcal{M}$ be the set of processes that take an infinite number of steps that eventually write M in registers. By Lemma 25 at least one process p_M that takes an infinite number of steps writes M forever in all registers. Before writing M in some register, p_M makes a $NBSscan$ of all registers returning snapshot V. By stability $V = M$ and by minimality of M, for each i we have $V[i] = M$. Then there is a time $\sigma \geq \tau'$ such that for each i, $1 \leq i \leq m$, we have $R[i]^\sigma = M$.

Assume by contradiction that there is some stable set X such that $X \cap M = \emptyset$. Let $\mathcal{X}$ be the set of such sets, and $\mathcal{P}$ be the set of processes that take an infinite number of steps and eventually write in the registers any $X \in \mathcal{X}$. From Lemma 23, if $card(\mathcal{P}) < m$, M will be forever in all registers but then $card(\mathcal{P}) = 0$ contradicting the hypothesis. Then we have $card(\mathcal{P}) \geq m$.

Consider any minimal stable set N in $\mathcal{X}$, by a symmetric argument we prove that if $\mathcal{Q}$ is the set of processes that take an infinite number of steps and eventually write stable sets Y such that $Y \cap N = \emptyset$ then $card(\mathcal{Q}) \geq m$.

As $card(\mathcal{Q}) \geq m$, $card(\mathcal{P}) \geq m$, $\mathcal{P}$ and $\mathcal{Q}$ are disjoint and $n < 2m$, we get a contradiction.

Hence if $n < 2m$ all stable sets contain M, proving that all *Deliver* eventually return sets all containing M. Moreover, by Lemma 26, there is a value in M, proving the convergence property.