



**HAL**  
open science

## Linear Space Bootstrap Communication Scheme

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

► **To cite this version:**

Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, Sergio Rajsbaum. Linear Space Bootstrap Communication Scheme. 2012. hal-00717235v1

**HAL Id: hal-00717235**

**<https://hal.science/hal-00717235v1>**

Preprint submitted on 16 Jul 2012 (v1), last revised 21 Aug 2012 (v2)

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Linear Space Bootstrap Communication Scheme

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 processor forever. By doubling the space complexity, using  $2n - 1$  rather than just  $n$  registers, processes can erase each other's write operation, just a bounded number of times, and consequently 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,  $p_1, \dots, p_n$ , communicate using Single-Writer Multi-Reader (SWMR) registers,  $R_1, \dots, R_n$ , 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. [5, 9] are known that reduce the name space to  $M(n) = 2n - 1$  (and this is the best that can be done, except for some exceptional values of  $n$  [11]). 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 was raised in the context of *adaptive* computation [6, 7, 22, 2, 3]. There,  $n$  is unknown, and moreover processors “arrive” and “depart.” Their answer is first to solve adaptive

renaming to allocate MWMR registers to processors. But that approach would yield  $\Theta(n^2)$  space complexity. The first simplest question we address here is: In the most favorable situation, when  $n$  is known and what we want to solve is a task, how much space do we need? Is solving renaming first unavoidable? using  $c$  MWMR registers [7, 22].

The start of this paper is to show that indeed it is possible to solve any read-write wait-free solvable task with  $n$  MWMR registers, which necessarily implies no hidden renaming solution as it provably requires an allocated space larger than  $n$  [ ].

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, it is not hard to prove that no such emulation exists on less than  $n$  MWMR registers.

As said, with  $n$  registers we can solve renaming, 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. We conjecture that  $2n - 1$  registers are the minimum possible for wait-free emulation.

An application of the non-blocking emulation is that one can run any SWMR wait-free algorithm that solves a *task* [17] on  $n$  MWMR registers. In particular, one can run directly a SWMR  $(2n - 1)$ -renaming algorithm such as the one of [9] on top of  $n$  MWMR registers. This is a significant improvement over the previous  $\Theta(n^2)$  space renaming algorithms, when that algorithm is instantiated to our context. Admittedly, that algorithm will actually use fewer than  $n^2$  registers but nevertheless since it is a priori unknown how the algorithm will evolve a preallocation of  $n^2$  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 incurs some cost in time: a process may not be able to produce an output value, until another process finishes the algorithm and exists the emulation. The wait-free emulation can be used to

solve renaming, and this allows to solve a task  $\frac{2}{n}$  times faster than the non-blocking emulation, at the cost of  $n - 1$  additional MWMR registers.

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.

Finally, the paper sketches how to import our wait-free simulation to a dynamic system of bounded concurrency  $n$  - 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  $n$ .

There is a long history of space complexity results, starting with the mutual exclusion lower bound of [10] and even before there was interest in this (see references herein). In this context, [23] considers renaming with the same motivation that we do, and shows that  $2\lceil \log n \rceil + 1$  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. [12, 13, 15, 22], but we are not aware of general emulations. In [20] there is a lower bound for non-blocking implementations, but it is  $n - 1$  registers. In special cases it can be beaten [4]. In [14] 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 [8, 18]. The processes have unique ids in  $\{1, \dots, N\}$ , with  $N \gg n$ . 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 [16]. 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 [16] 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  $WRITE(p, x)$  and all the processes may read the value in the SWMR register of  $p$  with  $READ(p)$ .

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 [19] and the progress condition is non-blocking (resp. wait free).

A regular SWMR register [21], 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

---

**function** *Collect*():

```

1  V = ∅
2  for all R in R
3      V = V ∪ read(R)
4  return V
```

main loop:

```

repeat forever
5      View = Collect() ∪ View ∪ input
6      Let Reg be some register in R
7      write(Reg, View)
```

---

Figure 1: Generic algorithm.

last value in the register before the concurrent writes. With regular SWMR registers it is possible to wait-free free implement atomic SWMR register [18, 21]. 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 of  $\mathcal{R}$  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  $R \in \mathcal{R}$ ,  $R^\tau$  denotes the value of register  $R$  at time  $\tau$ . Similarly,  $View_p^\tau$  denotes the value of variable *View* of  $p$  at time  $\tau$ .

We say that  $v$  is *eventually forever* in  $R$  if there is a time  $\tau$  such that for all time  $\tau' > \tau$ ,  $v \in R^{\tau'}$ . We have directly from the algorithm:

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

Following [10], we say that register  $R$  is *covered* by process  $p$  at some time  $\tau$  if the next register that  $p$  writes after time  $\tau$  is  $R$ : more precisely  $p$  *covers*  $R$  at time  $\tau$  if, at time  $\tau$ , 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 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  $\tau$  if all processes covering  $R$  at time  $\tau$  are going to write sets containing  $V$  in register  $R$ . If  $R$  is  $V$ -covered at time  $\tau$ , next writings contains  $V$  (by processes covering  $R$ ) or contains  $R^\tau$  (by processes not covering  $R$  that read  $R$  before writing). We have:

**Lemma 2** *If, at some time  $\tau$ , register  $R$  is  $V$ -covered at time  $\tau$  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  $\tau$ , we prove that  $R$  contains  $R^\tau \cap V$ . For  $k = 0$  it is clear. Consider any writing of  $R$  after time  $\tau$  and assume this writing is made, say, by process  $q$ . Consider the two cases:

- either  $q$  does not cover  $R$  at time  $\tau$  and then  $q$  makes a *Collect* before writing  $R$ . This *Collect* occurs after time  $\tau$  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  $\tau$  and  $V \subseteq \text{View}_p$  then  $q$  writes a superset of  $R^\tau \cap V$  in  $R$ .

□

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

**Lemma 3** *If  $R$  is not covered at time  $\tau$  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  $\tau$  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  $\tau$ ,  $\text{card}\{R \in \mathcal{R} | v \in R^\tau\} \geq n$ , 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  $\tau$  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  $\tau$  after which every set returned by any *deliver* contains  $v$ .

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  $\tau$  such that  $R[i]^\tau = V[i]$ . *NBScan()* is only assumed to be non-blocking. Implementation of such non-blocking snapshot

---

```

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

```

---

Figure 2: Broadcast with  $m$  MWMMR registers

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 SWMMR 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 SWMMR registers for  $n$  pro-

cesses using an array  $R$  of  $m$  MWMMR registers, if  $m \geq n$ .

---

```

Shared variable :
    array of  $m$  MWMMR-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 SWMMR registers for  $n$  processes.

In the following to distinguish between the writings of MWMMR registers and the emulation of SWMMR writings, we denote the first ones using lower case and the second ones with upper case. To make its  $k$ -WRITING, 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 WRITING 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.

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 WRITING of  $v$  *succeeds* at time  $\tau$  if after time  $\tau$ ,  $v$  is forever in some register. By extension, we say that the WRITING of  $v$  succeeds if there is a time at which the WRITING 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  $\tau$ , then  $v'$  succeeds at time  $\tau$  too.*

By Lemma 4 and the code of the algorithm:

**Lemma 8** *If the WRITING of  $v$  succeeds at time  $\tau$ , 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  $\tau$  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  $\tau$ ,  $v$  is in  $n$  registers, then the WRITING of  $v$  succeeds at time  $\tau$ .*

**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  $k$ th 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 WRITING 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 WRITING 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 WRITING is concurrent.  $\square$

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 WRITING 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 WRITING terminates.*

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

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

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 WRITING of  $v$  by a process  $p$  that takes an infinite number of steps does not terminate, then infinitely often some WRITINGS terminate.*

PROOF. By contradiction, assume that the WRITING of some process  $p$  that takes an infinite number of steps does not terminate and only a finite number of WRITINGS 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 WRITING terminates—a contradiction  $\square$

Lemmas 11 and 16 prove that the algorithm in Figure 3 is a non blocking emulation of regular SWMR registers for  $n$  processes from  $n$  MWMM-Registers. We now use the classical wait free transformation [18, 21] from regular to atomic registers to conclude:

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

## 5.2 Lower bound

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

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

PROOF. Consider a set of  $n$  processes  $p_1, \dots, p_n$ . By contradiction, assume we can emulate the SWMR system with  $(n - 1)$ -MWMM 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  $p_1, \dots, p_k$ .

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

**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  $\text{READ}(p_1)$ . Once  $p_1$  ends its  $\text{WRITE}$ ,  $p_n$  takes steps and does not find any value written by  $p_1$ , contradicting the semantics of a register.  $\square$

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  $\mathcal{R}_1 = \{R_1\}$ .

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

**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  $\text{READ}(p_{k+1})$ .  $p_{k+1}$  ends its WRITING and has only written registers in  $\mathcal{R}_k$ , then each process in  $\{p_1, \dots, p_k\}$  executes one step and overrides each register in  $\mathcal{R}_k$ . Then  $p_n$  executes the code for  $\text{READ}(p_{k+1})$ . But this execution is indistinguishable (for all processes different from  $p_{k+1}$ ) from one in which  $p_{k+1}$  does not make any WRITING and the  $\text{READ}$  may not return the value WRITTEN by  $p_{k+1}$ .  $\square$

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 MWMM registers. Now, consider process  $p_n$  and assume it runs the code  $\text{WRITE}(p_n, p_n)$ . Run  $e_n$  is an extension of  $e_{n-1}$  in which  $p_n$  takes steps until it ends its  $\text{WRITE}$  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  $\text{WRITE}(p_1, p_1)$ , at the end of this WRITING, the value WRITTEN by  $p_n$  is not in any MWMM 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 WRITING. Now if  $p_1$  runs  $\text{READ}(p_n)$ ,  $p_1$  cannot get the value written by  $p_n$ .  $\square$

## 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 [5, 9]. 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 a unique identity  $id(p)$  in the set  $\{1, \dots, 2n - 1\}$ . 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 a unique identity in the set  $\{1, \dots, n\}$ .

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 position  $p$  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 ar-

---

```

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( $p, x$ ):
1   $v = (x, p, k)$ 
2   $next = 0$ 
3   $nextReg = W[next]$ 
4   $View = View \cup \{v\}$ 
5   $Name = \text{index of } p \text{ in } View.proc$ 
6  do
7       $Snap = Scan()$ 
8       $View = Snap \cup View$ 
9      if ( $Snap[card(View.proc)] = \perp$ ) then
10          $Name = \text{index of } p \text{ in } View.proc$ 
11          $Update(PR[card(View.proc)], View)$ 
12     else
13          $Update(nextReg, View)$ 
14          $next = (next + 1) \bmod n$ 
15         if  $next = n - 1$  then  $nextReg = PR[Name]$ 
16         else  $nextReg = W[next]$ 
17 until ( $card\{r | v \text{ in } Snap[r]\} \geq n$ )
18  $k = k + 1$ 

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

```

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

ray  $W$  (Working registers) of  $n - 1$  MWMR-registers ( $W[0], \dots, W[n - 2]$ ) and an array PR (Personal Registers) of  $n + 1$  MWMR-registers ( $PR[0], \dots, PR[n]$ ).

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 per-

sonal register (Lines 5 and 10).

As usual to WRITE a value  $x$  (assuming it is the  $k$ th), 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 13) and in its personal register  $PR[Name]$  (Line 15). 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). If it not the case, the last non  $\perp$  cell in  $PR$  is before 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 WRITING terminates when  $v$  is in at least  $n$  registers (condition in Line 17): when  $v$  is in  $n$  registers at least one of these registers is not covered. Roughly speaking, WRITING is wait-free because eventually the last cell different 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 WRITING 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  $PR[0], \dots, PR[n - 1]$  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  $\tau$  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  $\tau$ .

Let  $p$  be a process that makes an infinite number of *Scan/Update*,  $p$  will make some *Scan* after time  $\tau$  getting  $PR[max]$ . Then there is a time  $\tau_p$  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  $\sigma$  be max of  $\tau_p$  for all  $p$  making an infinite number of *Scan/Update*, after time  $\sigma$  conditions (1) and (2) of the claim are satisfied.  $\square$

Let  $i$  such that  $0 \leq i \leq n$  and assume that  $PR[i]$  is written after time  $\sigma$ , 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  $\tau$ , 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 a unique index in *Participant*,  $q$  is the only writer of  $PR[i]$ .  $\square$

**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  $(v, p, k)$  is 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  $\tau$  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  $\tau$ , every process  $q$  that executes *Scan/update* reads  $PR[i]$  and includes it in its  $View_q$ . Then all processes that write after time  $\tau$  in some register will write  $v$  in this register too. As we assume that the WRITING 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 WRITING of  $v$  ends.  $\square$

Notice that the information in  $PR[n]$  can be easily integrated to  $PR[n-1]$ . So we can use only  $2n - 1$  MWMM registers. The shared MWMM 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 by:

---

```

9.1 if  $Name \neq \text{index of } p \text{ in } View.proc$  then
9.2    $Name = \text{index of } p \text{ in } View.proc$ 
9.3 if  $(card(View.proc) = n \text{ and } card(PR[n-1].proc) = n)$ 
   or  $(card(View.proc) < n \text{ and } Snap[card(View.proc)] = 1)$ 
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  $2n - 1$  MWMM registers for  $n$  processes.

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

## 7 $n$ -concurrency - sketch

We have seen how  $n$  processors emulate SWMM wait-free using  $2n - 1$  MWMM registers. What

if we have  $N$  processors  $N \gg n$  ( $N$  possibly infinite) but we have the notion of arrivals and departures of processors. Processors are to solve a task  $T$ . We have  $T$  defined over  $N$  processors. Yet we are to solve the task only under the condition that the number of processors that invoked the task (arrived) minus the number of processors that obtain an output (departed) is at any point of time less or equal to  $n$ . Suppose that we know that  $T$  is read-write wait-free solvable in the SWMM model under the assumption of  $n$ -concurrency. Can we solve  $T$  with just  $2n - 1$  MWMM registers, without indefinite postponement, i.e. the step complexity of a processor until it gets an output will be a function of  $n$  rather than  $N$ ?

In the full paper we show that under few additional observations outlined below the answer is positive.

The standard approach borrowed from the proof of the renaming algorithm in [?] is to assume there is an infinite postponement and arrive at contradiction. Indeed we are going to use the algorithm in [?] since in the SWMM model for  $N$  processor under bounded concurrency  $n$ , it does solve the following variant of renaming using  $M(n) = 2n - 1$  names. In this variant processors not only output a new name but also output a set of processors that were concurrent with them (loosely defined). Any two concurrent processors hold a distinct name.

Thus processors will be allocated a dedicated MWMM register according to the name they obtain solving  $n$ -concurrent renaming. Yet even to solve that a processor potentially may be forever postponed. To prevent it we make processors' behavior is controlled by two parameters: The maximum size concurrency they have observed in the system since their arrival. We notice that this value is monotonically non-decreasing. The other parameter is how early compared to other processors they arrived. A processor postponed forever will eventually have the maximum concurrency observed among all others eventually concurrent with it. Similarly, it will consider itself the oldest. If visible in the system (i.e. succeeded to appear in  $n$  registers) then all eventually will consider it the oldest.

If the oldest processor is invisible, then we can argue that its maximum concurrency observed is strictly larger than the one observed by any other processor eventually concurrent with it. If it is visible to others then all of them will consider it to be the oldest, and if they rank the older with smaller number, it will be the only one with rank 1 eventually. We now let processors which obtained a dedicated MWMR register to write their register and  $n - 1$  additional undedicated registers going from “left to right” while those that did not obtain yet a dedicated register writing  $n$  undedicated registers going from “right to left.” The starting position of these latter writes is controlled by a combination of rank and maximum concurrency observed. We are able to argue that eventually the starting location in which the oldest processor starts writing from “right to left” is not written by any other processor, hence it must progress - contradiction.

## 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. 10.1007/s00446-007-0041-1.
- [3] Marcos K. Aguilera. A pleasant stroll through the land of infinitely many creatures. *SIGACT News*, 35(2):36–59, June 2004.
- [4] 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.
- [5] 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.
- [6] Hagit Attiya and Arie Fouren. Polynomial 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.
- [7] Hagit Attiya and Arie Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31(2):642–664, February 2002.
- [8] Hagit Attiya and Jennifer Welch. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics*. John Wiley & Sons, 2004.
- [9] Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *PODC*, pages 41–51. ACM Press, 1993.
- [10] James E. Burns and Nancy A. Lynch. Bounds on shared memory for mutual exclusion. *Inf. Comput.*, 107(2):171–184, 1993.
- [11] Armando Castañeda and Sergio Rajsbaum. New combinatorial topology bounds for renaming: the lower bound. *Distributed Computing*, 22(5-6):287–301, 2010.
- [12] 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.
- [13] 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.
- [14] Faith Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5):843–862, September 1998.
- [15] 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.
- [16] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, January 1991.
- [17] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(2):858–923, 1999.
- [18] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
- [19] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
- [20] Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for nonblocking implementations. *SIAM J. Comput.*, 30(2):438–456, April 2000.
- [21] Leslie Lamport. On interprocess communication; part I and II. *Distributed Computing*, 1(2):77–101, 1986.
- [22] Mark Moir. Fast, long-lived renaming improved and simplified. *Sci. Comput. Program.*, 30(3):287–308, March 1998.
- [23] 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 *Delivers* 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  $\tau$ , let  $V \subseteq \bigcap_{1 \leq j \leq m} R[j]^\tau$  and let  $Q = \{q \in \Pi \mid V \not\subseteq \text{View}_q^\tau\}$ , if  $\text{card}(Q) < m$  then 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  $V \subseteq \text{View}_p$ .*

**PROOF.** If  $\text{card}(Q) < m$  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  $\text{View}_p$ .  $\square$

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  $\text{View}_p = V$ .*

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<sup>iii</sup> 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  $\tau$  after which we have forever  $E = \text{View}_p$ .*

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  $\tau_0$  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 *NBScan* 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  $\text{card}(\mathcal{P}) < m$ ,  $M$  will be forever in all registers but then  $\text{card}(\mathcal{P}) = 0$  contradicting the hypothesis. Then we have  $\text{card}(\mathcal{P}) \geq m$ .

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  $\text{card}(\mathcal{Q}) \geq m$ .

As  $\text{card}(\mathcal{Q}) \geq m$ ,  $\text{card}(\mathcal{P}) \geq m$ ,  $\mathcal{P}$  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.