Progress-Space Tradeoffs in Single-Writer Memory Implementations

Damien Imbs[*1], **Petr Kuznetsov**[*2], **and Thibault Rieutord**[*3]

1 **LIF, Aix-Marseille Université & CNRS, France, and Bremen University, Germany**
damien.imbs@lif.univ-mrs.fr
2 **LTCI, Télécom ParisTech, Université Paris Saclay, France**
petr.kuznetsov@telecom-paristech.fr
3 **LTCI, Télécom ParisTech, Université Paris Saclay, France**
thibault.rieutord@telecom-paristech.fr

Abstract

Many algorithms designed for shared-memory distributed systems assume the *single-writer multi-reader* (SWMR) setting where each process is provided with a unique register that can only be written by the process and read by all. In a system where computation is performed by a bounded number n of processes coming from a large (possibly unbounded) set of potential participants, the assumption of an SWMR memory is no longer reasonable. If only a bounded number of multi-writer multi-reader (MWMR) registers are provided, we cannot rely on an *a priori* assignment of processes to registers. In this setting, implementing an SWMR memory, or equivalently, ensuring *stable writes* (i.e., every written value persists in the memory), is desirable.

In this paper, we propose an SWMR implementation that adapts the number of MWMR registers used to the desired progress condition. For any given k from 1 to n, we present an algorithm that uses $n + k - 1$ registers to implement a *k-lock-free* SWMR memory. In the special case of 2-lock-freedom, we also give a matching lower bound of $n + 1$ registers, which supports our conjecture that the algorithm is space-optimal. Our lower bound holds for the strictly weaker progress condition of 2-obstruction-freedom, which suggests that the space complexity for k-obstruction-free and k-lock-free SWMR implementations might coincide.

1998 ACM Subject Classification C.2.4 Distributed Systems, F.1.1 Models of Computation.

Keywords and phrases Single-writer memory implementation, comparison-based algorithms, space complexity, progress conditions

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.9

1 Introduction

We consider a model of distributed computing in which at most n *participating* processes communicate via reading and writing to a shared memory. The participating processes come from a possibly unbounded set of *potential* participants: each process has a unique identifier (IP address, RFID, MAC address, etc.) which we, without loss of generality, assume to be an integer value. Given that processes do not have an *a priori* knowledge of the participating set, it is natural to assume that they can only *compare* their identifiers to establish their relative order, otherwise they essentially run the same algorithm [14]. This model is therefore

* This work has been supported by the Franco-German DFG-ANR Project DISCMAT (14-CE35-0010-02) devoted to connections between mathematics and distributed computing.

called *comparison-based* [2]. In the comparison-based model with bounded shared memory, we cannot assume that the processes are provided with a prior assignment of processes to distinct registers. The only suitable assumption, as is the case for anonymous systems [16], is that processes have access to *multi-writer multi-reader* registers (MWMR), without a prior assignment.

In this paper, we study *space complexity* of comparison-based algorithms implementing an *abstract* single-writer multi-reader (SWMR) memory. The abstract SWMR memory allows each participating process to *write* to a private abstract memory location and to *read* from the abstract memory locations of all other participating processes. The SWMR abstraction can be further used to build higher-level abstractions, such as renaming [2] and atomic snapshot [1].

To implement an SWMR memory, we need to ensure that every write performed by a participating process on its abstract SWMR register is *persistent*: every future abstract read must see the written value, as long as it has not been replace by a more recent *persistent* write. To achieve persistence in a MWMR system, the emulated abstract write may have to update *multiple* base MWMR registers in order to ensure that its value is not overwritten by other processes. A natural question arises: *How many base MWMR registers do we need?*

In this paper, we show that the answer depends on the desired progress condition. It is immediate that n registers are required for a *lock-free* implementation, i.e., we want to ensure that at least *one* correct process makes progress. Indeed, any algorithm using $n-1$ or less registers can be brought into the situation where *every* base register is *covered*, i.e., a process is about to execute a write operation on it [4]. If we let the remaining process p_i complete a new abstract write operation, the other $n-1$ processes may destroy the written value by making a *block write* on the covered registers (each covering process performs its pending write operation). Thus, the value written by p_i is "lost": no future read would find it. It has been recently shown that n base registers are not only necessary, but also sufficient for a lock-free implementation [8].

A *wait-free* SWMR memory implementation that guarantees progress to *every* correct process can be achieved with $2n-1$ registers [8]. The two extremes, lock-freedom and wait-freedom, suggest an intriguing question: is there a dependency between the amount of progress the implementation provides and its space complexity: if processes are guaranteed more progress, do they need more base registers?

Contributions. In this paper, we give an evidence of such a dependency. Using novel covering-based arguments, we show that any 2-*obstruction-free* algorithm requires $n+1$ base MWMR registers. Recall that k-obstruction-freedom requires that every correct process makes progress under the condition that at most k processes are correct [15]. The stronger property of k-*lock-freedom* [5] additionally guarantees that if more than k processes are correct, then at least k out of them make progress.

We also provide, for any $k = 1, \ldots, n$, a k-lock-free SWMR memory implementation that uses only $n+k-1$ base registers. Our lower bound and the algorithm suggest the following:

▶ **Conjecture 1.** *It is impossible to implement a k-obstruction-free SWMR memory in the n-process comparison-based model using $n+k-2$ MWMR registers.*

An interesting implication of our results is that 2-lock-free and 2-obstruction-free SWMR implementations have the same optimal space complexity. Given that n-obstruction-freedom and n-lock-freedom coincide with wait-freedom, we expect that, for all $k = 1, \ldots, n$, k-obstruction-free and k-lock-free (and all progress conditions in between [5]) require the

same number $n + k - 1$ of base MWMR registers. Curiously, our results highlight a contrast between complexity and computability, as we know that certain problems, e.g., consensus, can be solved in an obstruction-free way, but not in a lock-free way [11].

Related work. There has been a lot of work on space complexity in distributed computing, e.g., [12, 10, 17].

Delporte et al. [9] studied the space complexity of anonymous k-set agreement using MWMR registers, and showed a dependency between space complexity and progress conditions. In particular, they provide a lower bound of $n - k + m$ MWMR registers to solve anonymous *repeated* k-set agreement in the m-obstruction-free way, for $k < m$. Delporte et al. [7] showed that obstruction-free k-set agreement can be solved in the n-process comparison-based model using $2(n - k) + 1$ registers. This upper bound was later improved to $n - k + m$ for the progress condition of m-obstruction-freedom ($m \leq k$) by Bouzid, Raynal and Sutra [3]. In particular, their algorithm uses less than n registers when $m < k$.

To our knowledge, the only lower bound on the space-complexity of implementing an SWMR memory has been given by Delporte et al. [8] who showed that lock-free comparison-based implementations require n registers.

Delporte et al. [8] proposed two SWMR memory implementations: a lock-free one, using n registers, and a wait-free one, using $2n - 1$ registers. These algorithms are used in [6] to implement a *uniform* SWMR memory, i.e., assuming no prior knowledge on the number of participating processes. Assuming that p processes participate, the algorithms use $3p + 1$ and $4p$ registers for, respectively, lock-freedom and wait-freedom.

Roadmap. The paper is organized as follows. Section 2 defines the system model and states the problem. Section 3 presents a k-lock-free SWMR memory implementation. Section 4 shows that a 2-obstruction-free SWMR memory implementation requires $n + 1$ MWMR registers and hence that our algorithm is optimal for $k = 2$. Section 5 concludes the paper with implications and open questions. Secondary proofs, similar to those from [8], are delegated to the appendix.

2 Model

We consider the asynchronous shared-memory model, in whicha bounded number $n > 1$ of asynchronous crash-prone processes communicate by applying read and write operations to a bounded number m of *base* atomic multi-writer multi-reader atomic registers. An atomic register i can be accessed with two memory operations: $write(i, v)$ that replaces the content of the register with value v, and $read(i)$ that returns its content. The processes are provided with unique identifiers from an unbounded name space. Without loss of generality, we assume that the name space is the set of positive integers.

2.1 States, configurations and executions

An algorithm assigned to each process is a (possibly non-deterministic) automaton that accepts high-level operation requests as an application input. In each state, the process is poised to perform a *step*, i.e., a read or write operations on base registers. Once the step is performed, the process changes its state according to the result the step operation, possibly non-deterministically and possibly to a step corresponding to another high-level operation.

A *configuration*, or system state, consists of the state of all processes and the content of all MWMR registers. In the *initial* configurations, all processes are in their initial states, and all registers carry initial values.

We say that a step e by a process p is *applicable* to a configuration C, if e is the pending step of p in C, and we denote by Ce the configuration reached from C after p performed e. A sequence of steps $e_1, e_2, \ldots$ is applicable to C, if e_1 is applicable to C, e_2 is applicable to Ce_1, etc. A (possibly infinite) sequence of steps applicable to a configuration C is called an *execution from* C. A configuration C is said to be *reachable* from a configuration C', and denoted $C \in \mathit{Reach}(C')$, if there exists a finite execution α applicable to C', such that $C = C'\alpha$. If omitted, the starting configuration is the initial configuration, and we write $C \in \mathit{Reach}()$ if C is a reachable configuration for our algorithm.

Processes that take at least one step of the algorithm are called *participating*. A process is called *correct* in a given (infinite) execution if it takes infinitely many steps in that execution. Let $\mathit{Correct}(\alpha)$ denote the set of correct processes in the execution α.

2.2 Comparison-based algorithms

We assume that the processes are allowed to use their identifiers only to compare them with the identifiers of other processes: the outputs of the algorithm only depend on the inputs, the relative order of the identifiers of the participating processes, and the schedule of their steps. Formally, we say that an algorithm is *comparison-based*, if, for each possible execution α, by replacing the identifiers of participating processes with new ones preserving their relative order, we obtain a valid execution of the algorithm. Notice that the assumption does not preclude using the identifiers in communication primitives, it only ensures that decisions taken in the algorithm's run are taken only based on the their relative order of the identifiers.

In this model, m MWMR registers can be used to implement a wait-free m-component *multi-writer atomic-snapshot* memory [1]. The memory exports operations $\mathit{Update}(i, v)$ (updating position i of the memory with value v) and $\mathit{Snapshot}()$ (atomically returning the contents of the memory). In the comparison-based atomic-snapshot implementation, easily derived from the original one [1], $\mathit{Update}(i, v)$ writes only once, to register i, and $\mathit{Snapshot}()$ is read-only. For convenience, in our upper-bound algorithm we are going to use atomic snapshots instead of read-write registers.

2.3 SWMR memory

A single-writer multi-reader (SWMR) memory exports two operations: $\mathit{Write}()$ that takes a value as a parameter and $\mathit{Collect}()$ that returns a *multi-set* of values. It is guaranteed that, in every execution, there exists a reading map π that associates each complete Collect operation C, returning a multi-set $V = \{v_1, \ldots, v_s\}$, with a set of s Write operations $\{w_1, \ldots, w_s\}$ performed, respectively, by distinct processes $p_1, \ldots, p_s$ such that:

- The set $\{p_1, \ldots, p_s\}$ contains all processes that completed at least one write operation before the invocation of C;
- For each $i = 1, \ldots, s$, w_i is either the last write operation of process p_i preceding the invocation of C or a write operation of p_i concurrent with C.

Note that our definition does not guarantee atomicity of SWMR operations. Moreover, we do not require that processes are allocated with a unique MWMR register that can be used as a single writer register. Instead, we simply require that processes are able to simulate the use of single writer registers through implementing the SWMR memory.

Intuitively, a collect operation can be seen as a sequence of reads on *regular* registers [13], each associated with a distinct participating process. Such a collect object can be easily transformed into a *single-writer* atomic snapshot abstraction [1].

2.4 Progress conditions

In this paper we focus on two families of progress conditions, both generalizing the *wait-free* progress condition, namely *k-lock-freedom* and *k-obstruction-freedom*.

An execution α satisfies the property of *k-lock-freedom* [5] ($1 \leq k \leq n$) if at least $\min(k, Correct(\alpha))$ correct processes *make progress* in it, i.e., complete infinitely many high-level operations (in our case, Writes and Collects). The special case of n-lock-freedom is called *wait-freedom*. The property of *k-obstruction-freedom* [11, 15] requires that every correct process makes progress, under the condition that there are at most k correct processes. (If more than k processes are correct, no progress is guaranteed.)

In particular, k-lock-freedom is a stronger requirement than k-obstruction-freedom (strictly stronger for $1 \leq k < n$). Indeed, both require that every correct process makes progress when there are at most k correct processes, but k-lock-freedom additionally requires that some progress is made even if there are more than k correct processes.

3 Upper bound: k-lock-free SWMR memory with $n + k - 1$ registers

Consider a *full-information* algorithm in which every process alternates atomic snapshots and updates, where each update performed by a process incorporates the result of its preceding snapshot. Every value written to a register will *persist* (i.e., will be present in the result of every subsequent snapshot), unless there is another process poised to write to that register. The pigeonhole principle implies that k processes can cover at most k distinct registers at the same time. Thus, if, at a given point of a run, a value is present in n registers, then the value will persist. This observation implies a simple n-register *lock-free* SWMR implementation in which a high-level Write operation alternates snapshots and updates of all registers, one by one in the round-robin fashion, until the written high-level value is present in all n registers. A high-level Collect operation can simply return the set of the most recent values (defined using monotonically growing sequence numbers) returned by a snapshot operation. 0+0 The *wait-free* SWMR memory implementation in [8] using $2n - 1$ registers follows the n-register lock-free algorithm but, intuitively, for each participating process, replaces register n with register $n - 1 + pos$ where pos is the rank of the process among the currently observed participants. This way, there is a time after which every participating process has a dedicated register to write, and each value it writes will persist. In particular, every value it writes will be seen by all processes and will eventually be propagated to the $n - 1$ first registers.

To implement a k-lock-free SWMR memory using $n + k - 1$ registers, a process should determine, in a dynamic fashion, to which out of the last $k - 1$ registers to write. In our algorithm, by default, a Write operation only uses the first n registers, but if a process observes that its value is absent from some registers in the snapshot (some of its previous writes have been overwritten by other processes), it uses extra registers to propagate its value. The number of these extra registers depends on how many other processes have been observed as making progress.

Algorithm 1: k-lock-free SWMR implementation using $n+k-1$ MWMR registers.

```
1  View : list of triples of type (ValueType, IdType, ℕ), initially set to ∅;
2  opCounter ∈ ℕ, initially set to 0;

3  Write(v):
4      ActiveProcs = {id};
5      View = View ∪ (v, id, opCounter);
6      WritePos = 0;
7      WritePosMax = n;
8      do
9          Snap = MEM.snapshot();
10         ActiveProcs = ActiveProcs ∪ {pid, ∃(_, pid, c) ∈ Snap, ∀(_, pid, c') ∈ View, c > c'};
11         View = View ∪ Snap;
12         Update(MEM[WritePos], View);
13         WritePos = WritePos + 1 (mod WritePosMax);
14         WritePosMax = min(n + |ActiveProcs| − 1, n + k − 1);
15     while |{m ∈ {1, ..., n + k − 1}, (v, id, opCounter) ∈ Snap[m]}| < n;
16     opCounter = opCounter + 1;
17 End Write;

18 Collect():
19     Reads = MEM.snapshot();
20     V = ∅;
21     forall pid such that (_, pid, _) ∈ Reads do
22         V = V ∪ {v} with v such that (v, pid, max{c ∈ ℕ, (_, pid, c) ∈ Reads}) ∈ Reads;
23     Return V;
24 End Collect;
```

3.1 Overview of the algorithm

Our k-lock-free SWMR implementation, which uses $n+k-1$ base MWMR registers, is presented in Algorithm 1.

In a Write operation, the process adds the operation to be performed to its local view (line 5). The process then attempts to add its local view, together with the outcome of a snapshot, to each of the first *WritePosMax* registers, where *WritePosMax* is initially set to n and then adapts to the number of processes observed as concurrently active (lines 8–15). The writing process continues to do so until its Write operation value is present in at least n registers (line 15).

In this algorithm, the $k-1$ extra registers are used according to the liveness observed by blocked processses. In order to be allowed to use the last register, a process must fail to complete its write while observing at least $k-1$ other processes completing their own. This ensures that when a process access this last register, a k^{th} process is able to be observed by processes completing operations and thus will be helped to eventually complete.

The Collect operation is rather straightforward. It simply takes a snapshot of the memory and, for each participating process observed in the memory, it returns its most recent value (selected using associated sequence numbers, line 22).

3.2 Safety

At a high level, the safety of Algorithm 1 relies on the following property of register content stability:

▶ **Lemma 2.** *Let, at some point of a run of the algorithm, value (v, id, c) be present in some register r and such that no process is poised to execute an update on r (i.e., no process is between taking the snapshot of MEM (line 9) and the update of r (line 12)), then at all subsequent times $(v, id, c) \in r$, i.e., the value is present in the set of values stored in r.*

The persistence of the values in a specific uncovered register (Lemma 2) can be used to show the persistence of the value of a completed Write operation in *MEM*:

▶ **Lemma 3.** *If process p returns from a Write operation $(v, id(p), c)$ at time τ, then for any time $\tau' \geq \tau$ there is a register containing $(v, id(p), c)$.*

The proofs of the two above lemmas follow the path proposed in [8] and are delegated to the appendix.

With Lemma 3, we can derive the safety of our SWMR memory implementation (Section 2.3):

▶ **Theorem 4.** *Algorithm 1 safely implements an SWMR memory.*

Proof. It can be easily observed that a triplet (v, id, c) corresponds to a unique Write operation of a value v, performed by the process with identifier id. Therefore, a Collect operation returns a set of values proposed by Write operations from distinct processes, and thus the map π is well-defined.

By Lemma 3, the value (v, id, c) corresponding to a Write operation completed at time τ is present in some register r for any time $\tau' > \tau$, thus, the set of values resulting from any snapshot operation performed after time τ contains (v, id, c). Thus, for any complete Collect operation C, $\pi(C)$ contains a value for every process which completed a Write operation before C was invoked. Also, as each value returned by a Collect is the one associated to the greatest sequence number for a given process, it comes from the last completed Write or from a concurrent one. ◀

3.3 Progress

We will show, by induction on k, that Algorithm 1 satisfies k-lock-freedom. We first show, as in [8], that Write operations of Algorithm 1 are 1-lock-free (the proof is delegated to the appendix):

▶ **Lemma 5.** *Write operations in Algorithm 1 satisfy 1-lock-freedom.*

The induction step relies primarily on the helping mechanism. This mechanism guarantees that a process making progress eventually ensures that the processes it observes as having a pending operation also make progress (the mechanism is similar to the one of the wait-free implementation of [8]; the proof is also delegated to the appendix):

▶ **Lemma 6.** *If a process q performing infinitely many operations sees $(v, id(p), c)$, and if p is correct, then p eventually completes its c^{th} Write operation.*

By the base case provided by Lemma 5 and Lemma 6, we have:

▶ **Lemma 7.** *Write operations in Algorithm 1 satisfy k-lock-freedom.*

Proof. We proceed by induction on k, starting with the base case of $k = 1$ (Lemma 5). Suppose that Write operations satisfy ℓ-lock-freedom for some $\ell < k$. Consider a run in which at least $\ell + 1$ processes are correct, but only ℓ of them make progress (if such a run doesn't exist, the algorithm satisfies $(\ell + 1)$-lock-freedom). In this run, at least one correct process is eventually blocked in a Write operation. According to Lemma 6, the ℓ processes performing infinitely many Write operations eventually do not observe new values written by other processes. By the algorithm, these processes eventually never write to the last $k - \ell > 0$ registers.

A correct process that never completes a Write operation will execute the while loop (lines 8–15) infinitely many times, and thus, will infinitely often take a snapshot and update its local view (line 9). In particular, it will eventually observe a new Write operation performed by each of the ℓ processes completing infinitely many Write operations. It will then eventually include at least $\ell + 1$ processes in its set of active processes (i.e., the ℓ processes performing infinitely many Write operations and itself). It will therefore eventually write to the $(n + \ell)^{th}$ register infinitely often. In the considered run, this register is written infinitely often only by correct processes which do not complete new Write operations. The value from at least one of such process will then be observed by the ℓ processes making progress. By Lemma 6, this process will eventually complete its Write operation — a contradiction. ◀

Collect operations in Algorithm 1 clearly satisfy wait-freedom as there are no loops and MWMR snapshot operations are wait-free. Thus Lemma 7 and the wait-freedom of Collect operations imply:

▶ **Theorem 8.** *Algorithm 1 is a k-lock-free implementation of an SWMR memory for n processes using $n + k - 1$ MWMR registers.*

4 Lower bound: impossibility of 2-obstruction-free SWMR memory implementations with n MWMR registers

The algorithm in Section 3 gives an upper bound of $n+k-1$ on the number of MWMR registers required to implement an SWMR memory satisfying the k-lock-free progress condition in the comparison-based model. In this section, we present a lower bound on the number of MWMR registers required in order to provide a 2-obstruction-free, and hence also a 2-lock-free, SWMR memory implementation.

4.1 Overview of the lower bound

Our proof relies on the concepts of *covering* and *indistinguishability*.

A register is *covered* at a given point of a run if there is at least one process poised to write to it (we say that the process covers the register). Hence, a covered register cannot be used to ensure persistence of written data: by awakening the covering process, the adversarial scheduler can overwrite it. This property alone can be used to show that n registers are required for an obstruction-free (and hence also for a 1-lock-free) SWMR memory implementation [4], but not to obtain a lower bound of *more* than n shared resources as there is always one which remains uncovered.

Indistinguishability captures bounds on the knowledge that a process has of the rest of the system. Two system states are *indistinguishable* for a process if it has the same local state in both states and if the shared memory includes the same content. Thus, in an SWMR memory implementation, a Write operation can safely terminate only if, in all indistinguishable states, its value is present in a register that is not covered (by a process unaware of that value).

In our proof, we work with a composed notion of covering and indistinguishability. The idea is to show that there is a large set of reachable system states, indistinguishable to a given process p, in which different *sets of registers* are covered. Intuitively, if a set of registers is covered in one of these indistinguishable states, p must necessarily write to a register outside of this set in order to complete a new Write operation. Hence, if such indistinguishable states exist for *all* register subsets, then p must write its value to *all* registers. To perform infinitely many high-level Write operations, p must then write infinitely often to all available registers. But then any other process p' taking steps can be masked by the execution of p (i.e., any write p' makes to a MWMR register can be scheduled to be overwritten by p). This way we establish that no 2-obstruction free implementation exists, as it requires that at least two processes must be able to make progress concurrently.

4.2 Preliminaries

Assume, by contradiction, that there exists a 2-obstruction-free SWMR implementation using only n registers. To establish a contradiction, we consider a set of runs by a fixed set Π of n processes in which every process performs infinitely many Write operations with monotonically increasing arguments. Let $\mathcal{R}$ denote the set of n available registers.

Indistinguishability

A configuration C is said to be *indistinguishable* from a configuration C' for a set of processes P, if the content of all registers and the states of all processes in P are identical in C and C'. Given a set of configurations $\mathcal{D}$, let $I(\mathcal{D}, P)$ denote that any two configurations from $\mathcal{D}$ are indistinguishable for P.

We say that an execution is P-only, for a set of processes P, if it consists only of steps by processes in P. We say that a set of processes P is *hidden* in an execution α if all writes in α performed by processes in P are overwritten by some processes not in P, without any read performed by processes not in P in between. Given a sequence of steps α and a set of processes P, let $\alpha|_P$ be the sub-sequence of α containing only the steps from processes in P.

▶ **Observation 9.** *If a P-only execution α is applicable to a configuration C from a set of configurations $\mathcal{D}$ indistinguishable for P, i.e., $C \in \mathcal{D}$ and $I(\mathcal{D}, P)$, then α is applicable to any configuration $C' \in \mathcal{D}$, and it maintains the indistinguishability of configurations for P, i.e., $I(\{C'\alpha, C' \in \mathcal{D}\}, P)$.*

Let us denote as $\mathcal{D}\alpha$ the set of all configurations reached from $C \in \mathcal{D}$ by applying α. A similar observation can be made concerning hidden executions:

▶ **Observation 10.** *Given an execution α applicable to C with C from a set of configurations $\mathcal{D}$ indistinguishable for P, if processes in $\Pi \setminus P$ are hidden in α, then $\alpha|_P$ is applicable to any $C' \in \mathcal{D}$, and $I(\{C'\alpha|_P, C' \in \mathcal{D}\}, P)$.*

Coverings and confusion

We say that a set of processes P *covers* a set of registers R in some configuration C, if for each register $r \in R$, there is a process $p \in P$ such that the next step of p in C is a write on r (the predicate is denoted $Cover(R, P, C)$).

Our lower bound result relies on a concept that we call *confusion*. We say that a set of processes P are *confused* on a set of registers S in a set of reachable configurations $\mathcal{D}$, denoted $Confused(P, S, \mathcal{D})$, if and only if:

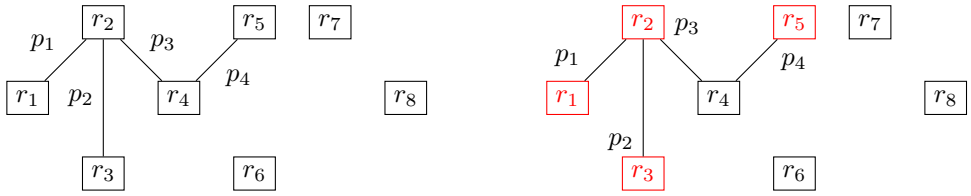

Figure 1 Processes $\{p_5, p_6, p_7, p_8\}$ are confused on registers $\{r_1, r_2, r_3, r_4, r_5\}$; an example of a possible covering is given on the right.

1. $I(\mathcal{D}, P)$.
2. $|S| + |P| = n + 1$.
3. For any process $p \in \Pi \setminus P$, there exist two registers $r_p, r'_p \in S$ such that, for any configuration $D \in \mathcal{D}$, there exists $D' \in \mathcal{D}$, such that p covers r_p in D and r'_p in D', or vice versa, and D and D' are indistinguishable to all other processes:

   $$\forall p \in \Pi \setminus P, \exists r_p, r'_p \in S, \forall D \in \mathcal{D}, \exists r \in \{r_p, r'_p\} :$$

   $$\mathit{Cover}(\{r\}, \{p\}, D) \wedge (\exists D' \in \mathcal{D}, I(\{D, D'\}, \Pi \setminus \{p\}) \wedge \mathit{Cover}(\{r_p, r'_p\} \setminus \{r\}, \{p\}, D')).$$

4. For any strict subset R of S, there exists $D \in \mathcal{D}$ such that R is covered by $\Pi \setminus P$ in D:

   $$\forall R \subsetneq S, \exists D \in \mathcal{D} : \mathit{Cover}(R, \Pi \setminus P, D).$$

Intuitively, processes in P are confused on S in $\mathcal{D}$, if $\mathcal{D}$ is a set of indistinguishable configurations for P, such that any *strict* subset of S is covered by $\Pi \setminus P$ in some configuration of $\mathcal{D}$ (Conditions 1 and 4). Additionally, we require that as much processes are confused as possible (Condition 2) and that the property holds for a set of configurations $\mathcal{D}$ where processes not in P might be covering only one out of 2 given registers and may be covering them independently of other processes states in $\mathcal{D}$ (Condition 3).

In Figure 1, we give an example of confusing set of configuration $\mathcal{D}$ for 8 processes and 8 registers. Processes $\{p_5, p_6, p_7, p_8\}$ are confused on registers $\{r_1, r_2, r_3, r_4, r_5\}$. Registers are represented as nodes, and pairs of registers that a process might be covering are represented as edges. The set of indistinguishable configurations $\mathcal{D}$ for $\{p_5, p_6, p_7, p_8\}$ are defined via composition of states for p_1, p_2, p_3 and p_4 in which they, respectively, cover registers in $\{r_1, r_2\}$, $\{r_2, r_3\}$, $\{r_2, r_4\}$ and $\{r_4, r_5\}$. An example of a covering of $\{r_1, r_2, r_3, r_5\}$ for some particular execution is presented on the right side of Figure 1.

First, we are going to provide an alternative property for Condition 4 of the definition of $\mathit{Confused}(P, S, \mathcal{D})$. The idea is that, given $(P, S, \mathcal{D})$ satisfying Conditions 1, 2 and 3, Condition 4 is satisfied if and only if the graph induced by the sets of registers that may be covered by processes in $\Pi \setminus P$ (as represented in Figure 1) forms a connected component over S. More formally, that Condition 4 is satisfied if and only if, for any partition of S into two non-empty subsets S_1 and S_2, there is a process in $\Pi \setminus P$ for which the set of two registers it may be covering in $\mathcal{D}$ intersects with both S_1 and S_2:

▶ **Lemma 11.** *$\forall P \subseteq \Pi, \forall S \subseteq \mathcal{R}, \forall \mathcal{D} \subseteq \mathit{Reach}()$ satisfying Conditions 1, 2 and 3 of the confusion definition, we have $\forall R \subsetneq S, \exists D \in \mathcal{D} : \mathit{Cover}(R, \Pi \setminus P, D)$ if and only if:*

$$\forall S_1, S_2 \subseteq S, (S_1 \neq \emptyset \wedge S_2 \neq \emptyset \wedge S_1 \cup S_2 = S \wedge S_1 \cap S_2 = \emptyset) :$$

$$\exists r_1 \in S_1, r_2 \in S_2, p \in \Pi \setminus P, D_1, D_2 \in \mathcal{D} : (\mathit{Cover}(\{r_1\}, \{p\}, D_1) \wedge \mathit{Cover}\left((\{r_2\}, \{p\}, D_2)\right).$$

Proof. Let us fix some $P \subseteq \Pi$, $S \subseteq \mathcal{R}$, and $\mathcal{D} \subseteq \mathit{Reach}()$ satisfying Conditions 1, 2 and 3 of the confusion definition.

First, let us assume that Condition 4 is also satisfied and consider any partition of S into non-empty subsets S_1 and S_2 (i.e., $S_1 \neq \emptyset$, $S_2 \neq \emptyset$, $S_1 \cap S_2 = \emptyset$ and $S_1 \cup S_2 = S$). Assume now that there does not exist any process $p \in \Pi \setminus P$ such that p might be covering a register from S_1 or a register from S_2 in $\mathcal{D}$. This implies that processes in $\Pi \setminus P$ can be partitioned into two subsets Q_1 and Q_2 (with $Q_1 \cap Q_2 = \emptyset$ and $Q_1 \cup Q_2 = \Pi \setminus P$) such that processes in Q_1, respectively Q_2, may cover registers from S_1, respectively S_2, in $\mathcal{D}$. By construction of the partitions, we have $|Q_1| + |Q_2| = |\Pi \setminus P|$ and $|S_1| + |S_2| = |S|$. Using the fact that Condition 2 is satisfied by P and S we obtain from $|S| + |P| = n + 1$ that $|S_1| + |S_2| + (n - (|Q_1| + |Q_2|)) = n + 1$, and thus, that $|S_1| + |S_2| = |Q_1| + |Q_2| + 1$. This implies that either $|Q_1| < |S_1|$ or $|Q_2| < |S_2|$, w.l.o.g., let $|Q_1| < |R_1|$. Now consider $r \in S_2$, $S \setminus \{r\}$ is a strict subset of S, and therefore Condition 4 implies that there exists $D \in \mathcal{D}$ such that $\mathit{Cover}(S \setminus \{r\}, \Pi \setminus P, D)$. As registers in S_1 can only be covered by processes from Q_1, then we have $\mathit{Cover}(S_1, Q_1, D)$. Recall that, by the pigeonhole principle, a set of processes cannot cover more registers than processes it contains. But $|Q_1| < |R_1|$ — a contradiction.

Now let us assume that given any partition of S into non-empty subsets S_1 and S_2, there exists a process $p \in \Pi \setminus P$ such that p might be covering a register in S_1 or a register in S_2 in $\mathcal{D}$. Let us show that any strict subset R of S is covered in some configuration from $\mathcal{D}$ and, hence, that Condition 4 is satisfied. The idea consists in inductively selecting a subset of the configurations in $\mathcal{D}$ by fixing a process in $\Pi \setminus P$ to cover a new register of R. The trick is to select a process which remains with a single choice if it wants to cover a register not yet covered in all remaining configurations by other processes.

Let S_0 be a non-empty subset of S. Let p_0 be a process from $\Pi \setminus P$ which might be covering a register r_0 in S_0 or a register r'_0 in $S \setminus S_0$ in $\mathcal{D}$. Let us assume that such a process exists and consider $\mathcal{D}_0$ to be the subset of $\mathcal{D}$ including all configurations in which p_0 is covering r'_0. Now let $S_1 = S_0 \cup \{r'_0\}$ and repeat this process by selecting some p_1 and computing $\mathcal{D}_1$ using the same procedure, etc... As long as a process can be selected satisfying the condition, the sets S_i keeps increasing with i. Consider the round j at which the procedure fails to find such a process. This implies that there is no process which might be covering a register from either S_j or $S \setminus S_j$ in $\mathcal{D}_{j-1}$. Note that by construction $\mathcal{D}_{j-1}$ is a non-empty subset of $\mathcal{D}$.

If $S_j \neq S$, then S_j and $S \setminus S_j$ forms a partition of S into two non-empty subsets. Thus, by assumption, there exists a process q which might be covering a register r_q in S_j or a register r'_q in $S \setminus S_j$ in $\mathcal{D}$. Consider some configuration $D \in \mathcal{D}_{j-1}$. According to Condition 3 of the confusion definition, as $D \in \mathcal{D}$, q is covering either r_q or r'_q in D, and there exists a configuration D' in which q is covering the other register in $\{r_q, r'_q\}$, relatively to D, and such that D and D' are indistinguishable to all other processes. As in D and D', all processes except q are in the same local state, either $D' \in \mathcal{D}_{j-1}$ or else q was a process selected in some early iteration. But if q was selected in an earlier iteration, both r_q and r'_q would be included in S_j. Thus D and D' belong to $\mathcal{D}_{j-1}$ and therefore q is a valid selection for p_j. This contradiction implies that therefore $S_j = S$.

By construction, all registers in $S_j \setminus S_0$ are covered in all configurations in $\mathcal{D}_{j-1}$. As this is true for any non-empty S_0 and as $S_j = S$, any strict subset of S is covered in some configuration of $\mathcal{D}$ and therefore Condition 4 is satisfied. ◀

The alternative formulation of Condition 4 provided by Lemma 11 can be used to show that, given any confusion for some non-empty P, S and $\mathcal{D}$, we can identify a process $p \in \Pi \setminus P$ and a register $r \in S$ such that $P \cup \{p\}$ is confused on $S \setminus \{r\}$ for a subset $\mathcal{D}'$ of $\mathcal{D}$:

▶ **Lemma 12.** *Given $P \subsetneq \Pi$, $S \subseteq \mathcal{R}$ and $\mathcal{D} \subseteq Reach()$:*
$Confused(P, S, \mathcal{D}) \implies (\exists p \in \Pi \setminus P, \exists r \in S, \exists \mathcal{D}' \subseteq \mathcal{D} : Confused(P \cup \{p\}, S \setminus \{r\}, \mathcal{D}'))$.

Proof. Note that, according to Condition 3, a process may be covering exactly two registers from S in $\mathcal{D}$, thus, the sum of how many distinct processes may be covering each register in S equals $2|\Pi \setminus P| = 2(n - |P|)$. But any register $r \in S$ must be potentially covered at least by one process in $\Pi \setminus P$ in $\mathcal{D}$ as any strict subset of S may be covered (Condition 4). Therefore, there exists a register $r_c \in S$ which can be covered by a single process $p_c \in \Pi \setminus P$ in $\mathcal{D}$. Indeed, if all registers in S might be covered by two distinct processes then the sum, equal to $2(n - |P|)$, would be greater than or equal to $2|S|$. This is not possible as according to Condition 2, $|P| + |S| = n + 1$ and thus $n - |P| < |S|$.

Let us fix some $D_c \in \mathcal{D}$ and let $\mathcal{D}_c$ be the subset of $\mathcal{D}$ which includes all configurations in $\mathcal{D}$ that are indistinguishable to p_c from D_c. Let us show that $Confused(P \cup \{p_c\}, S \setminus \{r_c\}, \mathcal{D}_c)$. Condition 1 holds as by construction all configurations in $\mathcal{D}_c$ are indistinguishable to p_c and as they are indistinguishable to all processes in P, since $\mathcal{D}_c$ is a subset of $\mathcal{D}$. It is immediate, as we add a register and remove a process, that Condition 2 holds.

Now consider any process $p \in \Pi \setminus (P \cup \{p_c\})$ and any configuration $D \in \mathcal{D}_c$. As $p \in \Pi \setminus P$ and $D \in \mathcal{D}$, Condition 3 of the confusion definition implies that that there exists $D' \in \mathcal{D}$, $I(\{D, D'\}, \Pi \setminus \{p\})$, such that p covers r_p and r'_p in D and D' (respectively of vice-versa). Since $I(\{D, D'\}, \Pi \setminus \{p\})$, $D' \in \mathcal{D}_c$, and since p_c is the only process which may cover r_c in $\mathcal{D}$, r_p and r'_p belong to $S \setminus \{r_c\}$. Thus Condition 3 is verified for $P \cup \{p\}$, $S \setminus \{r_c\}$ and $\mathcal{D}_c$.

Lastly, let us consider some partition of $S \setminus \{r_c\}$ into two non-empty subsets S_1 and S_2. Both $(S_1 \cup \{r_c\}, S_2)$ and $(S_1, S_2 \cup \{r_c\})$ form a partition of S in two non-empty subsets. Thus, as $Confused(P, S, \mathcal{D})$, we can apply Lemma 11 and obtain that $\exists p_1, p_2 \in \Pi \setminus P$ such that p_1, respectively p_2, might cover registers from either $S_1 \cup \{r_c\}$ or S_2, respectively either S_1 or $S_2 \cup \{r_c\}$, in $\mathcal{D}$. It follows that p_c cannot be both p_1 and p_2 as p_c might cover only two registers in $\mathcal{D}$, one of which is r_c. Thus, depending whether the other register belongs to S_1 or S_2, p_1 or p_2 is disctinct from p_c. W.l.o.g, assume that it is p_1. As p_c is the only process which may be covering r_c, this implies that p_1 might be covering a register from either S_1 or S_2. Furthermore, since Conditions 1, 2 and 3 applies to $P \cup \{p\}$, $S \setminus \{r_c\}$ and $\mathcal{D}_c$, we can apply Lemma 11 to obtain that Condition 4 is also verified. ◀

We now show that the characterization can be used to increase the number of registers that processes are confused on, by decreasing the number of confused processes:

▶ **Lemma 13.** *Let $P \subsetneq \Pi$, $S \subseteq \mathcal{R}$ and $\mathcal{D} \subset Reach()$ such that $Confused(P, S, \mathcal{D})$.*
Given $C \in \mathcal{D}$, if $\exists p \in P, r_1 \in S, r_2 \in \mathcal{R} \setminus S$ and if there exists P-only executions α_1 and α_2, applicable to C such that $I(\{C\alpha_1, C\alpha_2\}, \Pi \setminus \{p\})$ and such that $Cover(\{r_1\}, \{p\}, C\alpha_1)$ and $Cover(\{r_2\}, \{p\}, C\alpha_2)$, then we have $Confused(P \setminus \{p\}, S \cup \{r_2\}, (\mathcal{D}\alpha_1) \cup (\mathcal{D}\alpha_2))$.

Proof. Following Observation 9, as α_1 and α_2 are P-only, and since $\mathcal{D}$ satisfies Condition 1 for P, $(\mathcal{D}\alpha_1) \cup (\mathcal{D}\alpha_2)$ satisfies Condition 1 for $P \setminus \{p\}$. Condition 2 trivially holds since a process is removed from P and a register is added to S.

Condition 3 is satisfied for all processes in $\Pi \setminus P$ and configurations in $(\mathcal{D}\alpha_1) \cup (\mathcal{D}\alpha_2)$ as α_1 and α_2 are P-only, and since $\mathcal{D}$ satisfies Condition 3 for any process in $\Pi \setminus P$. As configurations in $\mathcal{D}$ are indistinguishable to p, p may only cover r_1 if $D \in \mathcal{D}\alpha_1$ and cover r_2 if $D \in \mathcal{D}\alpha_2$. But as given any $D \in \mathcal{D}$ we have $I(\{D\alpha_1, D\alpha_2\}, \Pi \setminus \{p\})$ (as $I(\mathcal{D}, P)$ following Observation 9), Condition 3 is also satisfied for p.

Since Consition 1, 2 and 3 are satisfied, we can thus apply Lemma 11 to obtain that $Confused(P \setminus \{p\}, S \cup \{r_2\}, (\mathcal{D}\alpha_1) \cup (\mathcal{D}\alpha_2))$. Indeed, a partition of two non-empty subsets of $S \cup \{r_2\}$ can be reduced to a partition of two non-empty subsets of S unless it is the partition S and $\{r_2\}$. But p may cover either $r_1 \in S$ or r_2. ◀

4.3 The lower bound

To establish our lower bound, we show that there is a set of reachable configuration $\mathcal{D}$ in which there is a process confused on all n registers. Intuitively, we proceed by induction on the number of "confusing" registers. For the base case, we show that the initial configuration can lead to a confusion of all but one process on *two registers*:

▶ **Lemma 14.** $\exists \mathcal{D} \in \mathit{Reach}(), \exists p \in \Pi, \exists S \subseteq \mathcal{R} : \mathit{Confused}(\Pi \setminus \{p\}, S, \mathcal{D})$.

Proof. Consider any two processes p_1 and p_2. Since the algorithm is comparison-based, the first write the two processes perform in a solo execution is on the same register, let us call it r. Let p_1 execute solo until it is about to write to r and then do the same with p_2, let C_1 be the resulting configuration. Consider the execution α from C_1 in which p_1 executes until it is poised to write to a register $r' \neq r$ and then p_2 executes its pending write on r. This execution is valid as p_1 must eventually write to an uncovered register.

As p_1 is hidden in α, $C_1\alpha$ and $C_1\alpha|_{\{p_2\}}$ are indistinguishable for all processes except p_1 (Observation 10). In $C_1\alpha$, p_1 is poised to write on r', but in $C_1\alpha|_{\{p_2\}}$, p_1 is poised to write on r, thus we have $\mathit{Confused}(\Pi \setminus \{p_1\}, \{r, r'\}, \{C_1\alpha, C_1\alpha|_{\{p_2\}}\})$. ◀

We now prove our inductive step. Given a set of configurations in which a set P of processes is confused on a set S of registers, we can obtain a set of configurations in which a set P' of processes are confused on a set S' of strictly more than $|S|$ registers:

▶ **Lemma 15.** $\exists \mathcal{D} \subseteq \mathit{Reach}(), P \subsetneq \Pi, S \subsetneq \mathcal{R} : \mathit{Confused}(P, S, \mathcal{D})$
$\implies \exists \mathcal{D}' \subseteq \mathit{Reach}(), P' \subseteq \Pi, S' \subseteq \mathcal{R}, S \subsetneq S' : \mathit{Confused}(P', S', \mathcal{D}')$.

Proof. Given $\mathit{Confused}(P, S, \mathcal{D})$, consider $C \in \mathcal{D}$ such that exactly $|S| - 1$ registers in S are covered by processes in $\Pi \setminus P$. Then we can reach a configuration in which all registers not in S are covered by processes in P. Indeed, when executed solo starting from C, a process must eventually write to a register that is not covered in C. Thus, it must eventually write either to a register in $\mathcal{R} \setminus S$ or to the uncovered register in S. Recall that, as $|S| + |P| = n + 1$, we have $|\mathcal{R} \setminus S| = |P| - 1$. Thus, by concatenating solo executions of processes in P until they are poised to write to uncovered registers, we reach a configuration $C\alpha$ in which *all* registers are covered, and let p be the process in P covering a register in S. Note that, as α is P-only, we have $\mathit{Confused}(P, S, \mathcal{D}\alpha)$. Thus:

$$\mathit{Cover}(\mathcal{R} \setminus S, P \setminus \{p\}, C\alpha) \wedge \mathit{Confused}(P, S, \mathcal{D}\alpha\}).$$

Now from this set of configurations, we are going to build a new one in which P is confused on *two* distinct sets of registers. By Lemma 12, there exist $p_c \in \Pi \setminus P$, $r \in S$ and $\mathcal{D}' \subseteq \mathcal{D}$ such that $\mathit{Confused}(P \cup \{p_c\}, S \setminus \{r\}, \mathcal{D}')$. Note that in $\mathcal{D}'$, the state of p_c can be chosen to be the state from any configuration from $\mathcal{D}$, and as we have $\mathit{Confused}(P, S, \mathcal{D})$ there must exists a configuration $C' \in \mathcal{D}$ in which p_c covers a register $r_c \in S \setminus \{r\}$. Let us select $\mathcal{D}'$ such that $C' \in \mathcal{D}'$.

If p is executed solo from $C'\alpha$, it must write infinitely often to *all* registers in S to ensure that it writes to an uncovered register. In a $\{p, p_c\}$-only execution from $C'\alpha$, p_c can hidden for arbitrarly many steps as long as p_c does not write to a register outside of S. But, as the algorithm satisfies 2-obstruction-freedom, p_c *must* eventually write to a register outside of S in such an execution. Consider the $\{p, p_c\}$-only execution β from $C'\alpha$ in which p_c is hidden and such that p_c execute until it is poised to write to some register $r' \in \mathcal{R} \setminus S$. Thus, we get two configurations $C'\alpha\beta$ and $C'\alpha\beta|_P$,indistinguishable to all processes but p_c, in which p_c covers, respectively, $r' \in \mathcal{R} \setminus S$ and $r_c \in S$. Thus, the conditions of Lemma 13 hold for $\mathcal{D}'$,

p_c, $\alpha\beta$ and $\alpha\beta|_{\{p\}}$ and so we obtain $\mathit{Confused}(P, (S \cup \{r'\}) \setminus \{r\}, (\mathcal{D}'\alpha\beta) \cup (\mathcal{D}'\alpha\beta|_{\{p\}}))$. As β is $\{p, p_c\}$-only and p_c is hidden in it, we have:

$$\mathit{Cover}(\mathcal{R} \setminus S, P \setminus \{p\}, C\alpha\beta) \wedge \mathit{Confused}(P, S, \mathcal{D}\alpha\beta|_{\{p\}}) \wedge$$

$$\mathit{Confused}(P, S \cup \{r'\} \setminus \{r\}, (\mathcal{D}'\alpha\beta) \cup (\mathcal{D}'\alpha\beta|_{\{p\}})).$$

Moreover, all configurations in the formula above are indistinguishable to processes in P, because $\alpha\beta$ is $P \cup \{p_c\}$-only and p_c is hidden in it (Observation 10).

Let $p' \in P$ be the process that covers r' in $C\alpha\beta$. According to p or p', every proper subset of S or $S \cup \{r'\} \setminus \{r\}$ may be covered at the same time by processes in $\Pi \setminus (P \cup \{p, p\})$, and all other registers are covered by processes in $P \setminus \{p, p'\}$. Thus, from $C\alpha\beta$, to complete a Write, p or p' must write to *all* registers in one of the sets S, $S \cup \{r'\} \setminus \{r\}$ or $\{r, r'\}$.

Consider any $\{p, p'\}$-only extension of $C\alpha\beta$. If one of $\{p, p'\}$ covers a register in $S \setminus \{r\}$, r or r', then the other process, in any solo extension, must write respectively to all registers in $\{r, r'\}$, S or $(S \cup \{r'\}) \setminus \{r\}$. In particular, since p' covers r' in $C\alpha\beta$, p running solo from $C\alpha\beta$ must eventually cover a register in $S \setminus \{r\}$ (as $|S| > 1$, since $|P| < n$ and $|P| + |S| = n + 1$). Then p' executing solo afterwards must write to r and r'. Let us stop p' when it covers a register $r'' \neq r$ for the last time before writing to r. Let γ be the resulting execution, and $E = C\alpha\beta\gamma$ be the resulting configuration. Let $\mathcal{E}$ and $\mathcal{E}'$ denote the sets of configurations indistinguishable from E to P defined as $\mathcal{D}\alpha\beta|_{\{p\}}\gamma$ and $(\mathcal{D}'\alpha\beta\gamma) \cup (\mathcal{D}'\alpha\beta|_{\{p\}}\gamma)$ respectively.

Now the following two cases are possible:

1. $r'' \notin S \cup \{r'\}$: In this case, we let p continue until it is poised to write on r, and then, we let the process in $P \setminus \{p, p'\}$ which covers r'' to proceed to its write. Let δ be this $\{p, p'\}$-only execution from E. As p' covers $r \in S$ in $E\delta$ and $r'' \in \mathcal{R} \setminus S$ in $E\delta|_{P \setminus \{p'\}}$, as $I(\{E\delta, E\delta|_{P \setminus \{p'\}}\}, \Pi \setminus \{p'\})$, and as $\mathit{Confused}(P, S, \mathcal{E})$, we can apply Lemma 13 and obtain $\mathit{Confused}(P \setminus \{p'\}, S \cup \{r''\}, (\mathcal{E}\delta) \cup (\mathcal{E}\delta|_{P \setminus \{p'\}}))$.
2. $r'' \in S \cup \{r'\}$, and so $r'' \in (S \cup \{r'\}) \setminus \{r\}$: Then we have the following sub-cases:
   - Some step performed by p in its solo execution from E makes p' to choose a register other than r to perform its next write in its solo extension. Clearly, this step of p is a write. From the configuration in which p is poised to execute this "critical" write, let p' run solo until it is poised to write to r and then let p complete its pending write. Let $E\delta$ be the resulting configuration.

     Now consider the execution in which p completes its "critical" write, then p' runs solo until it covers a register $r''' \neq r$. Let $E\delta'$ be the resulting configuration. Note that the states of the memory in $E\delta$ and $E\delta'$ are identical. Thus, $I(\{E\delta, E\delta'\}, \Pi \setminus \{p'\})$. Note that δ and in δ' are $\{p, p'\}$-only executions, and that p' covers r in $E\delta$ and r''' in $E\delta'$.
     a. If $r''' \in S$, as we have $\mathit{Confused}(P, (S \cup \{r'\}) \setminus \{r\}, \mathcal{E}')$, applying Lemma 13, we obtain $\mathit{Confused}(P \setminus \{p'\}, (S \cup \{r'\}), (\mathcal{E}'\delta) \cup (\mathcal{E}'\delta'))$.
     b. If $r''' \in \mathcal{R} \setminus S$, as we have $\mathit{Confused}(P, S, \mathcal{E})$, applying Lemma 13, we obtain $\mathit{Confused}(P \setminus \{p'\}, (S \cup \{r'''\}), (\mathcal{E}\delta) \cup (\mathcal{E}\delta'))$.
   - Otherwise, no write of p is "critical", and we let it run from E until it covers r (recall that, as p' covers $r'' \in (S \cup \{r'\}) \setminus \{r\}$, p must eventually write to all registers in S or $\{r, r'\}$ and, thus, to r). Let then p' run until it covers r and let δ be the resulting execution. From $E\delta$, let p' run until it becomes poised to write to a register $r''' \neq r$ for the first time, and then let p perform its pending write on r. Let λ be this extension. Note that as p' is hidden in λ, we have $I(\{E\delta\lambda, E\delta\lambda|_{\{p'\}}\}, \Pi \setminus \{p\})$. Also, δ and λ are $\{p, p'\}$-only executions such that p' covers r in $E\delta\lambda|_{\{p\}}$ and covers r''' in $E\delta\lambda$.
     a. If $r''' \in S$, as we have $\mathit{Confused}(P, (S \cup \{r'\}) \setminus \{r\}, \mathcal{E}')$, applying Lemma 13, we obtain $\mathit{Confused}(P \setminus \{p'\}, (S \cup \{r'\}), (\mathcal{E}'\delta\lambda) \cup (\mathcal{E}'\delta\lambda|_{\{p\}}))$.

b. If $r''' \in \mathcal{R} \setminus S$, as we have $\mathit{Confused}(P, S, \mathcal{E})$, applying Lemma 13, we obtain $\mathit{Confused}(P \setminus \{p'\}, (S \cup \{r'''\}), (\mathcal{E}\delta\lambda) \cup (\mathcal{E}\delta\lambda|_{\{p\}}))$. ◀

Our lower bound directly follows from Lemmata 14 and 15:

▶ **Theorem 16.** *Any n-process comparison-based 2-obstruction-free SWMR memory implementation requires $n + 1$ MWMR registers.*

Proof. By contradiction, suppose that an n-register algorithm exists. We show, by induction, that there is a reachable configuration in which a process is confused on *all* registers. Lemma 14 shows that there exists a reachable configuration in which $n - 1$ processes are confused on two registers. We can therefore apply Lemma 15 and obtain a configuration with a confusion with strictly more registers. By induction, there exist then a set of configurations $\mathcal{D}$ and $p \in \Pi$ such that $\mathit{Confused}(\{p\}, \mathcal{R}, \mathcal{D})$.

Thus, any strict subset of $\mathcal{R}$ is covered by the remaining $n - 1$ processes in some configuration in the (indistinguishable for p) set of configurations $\mathcal{D}$. But a process p may complete a Write operation if and only its write value is present in a register which is not covered (by a process not aware of the value) in any of the configurations indistinguishable to p. Therefore, in an infinite solo execution of p, p must write infinitely often to *all* registers. But then any arbitrarily long execution by any other process can be hidden by incorporating sufficiently many steps of p , violating 2-obstruction-freedom—a contradiction. ◀

5 Concluding remarks

This paper shows that the optimal space complexity of SWMR implementations depends on the desired progress condition: lock-free algorithms trivially require n registers, while 2-obstruction-free ones (and, thus, also 2-lock-free ones) require $n + 1$ registers. We also extend the upper bound to k-lock-freedom, for all $k = 1, \ldots, n$, by presenting a k-lock-free SWMR implementation using $n + k - 1$ registers. A natural conjecture is that the algorithm is optimal, i.e., no such algorithm exists for $n + k - 2$ registers for all $k = 1, \ldots, n$. Since for $k = 1$, 2 and n, k-obstruction-freedom and k-lock-freedom impose the same space complexity, it also appears natural to expect that this is also true for all $k = 1, \ldots, n$.

An interesting corollary to our results is that to implement a 2-obstruction-free SWMR memory we need strictly more space than to implement a 1-lock-free one. But the two properties are, in general, incomparable: a 2-solo run in which only one process makes progress satisfies 1-lock-freedom, but not 2-obstruction-freedom, and a run in which 3 or more processes are correct but no progress is made satisfies 2-obstruction-freedom, but not 1-lock-freedom. The relative costs of incomparable progress properties, e.g., in the (ℓ, k)-freedom spectrum [5], are yet to be understood.

An SWMR memory can be viewed as a *stable-set* abstraction with a conventional *put/get* interface: every participating process can put values to the set and get the set's content, and every get operation returns the values previously put. For the stable-set abstraction, we can extend our results to the *anonymous* setting, where processes are not provided with unique identifiers. Indeed, we claim that the same algorithm may apply to the stable-set abstraction for anonymous systems when the number of participating processes n is known. But the question of whether an *adaptive* solution exists (expressed differently, a solution that does not assume any upper bound on the number of participating processes) for anonymous systems remains open.

References

1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993. doi:10.1145/153724.153741.

2 Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–548, 1990. doi:10.1145/79147.79158.

3 Zohir Bouzid, Michel Raynal, and Pierre Sutra. Anonymous obstruction-free (n, k)-set agreement with n-k+1 atomic read/write registers. In *19th International Conference on Principles of Distributed Systems*, OPODIS '15, pages 18:1–18:17, 2015.

4 James E Burns and Nancy A Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, 1993.

5 Victor Bushkov and Rachid Guerraoui. Safety-liveness exclusion in distributed computing. In *34th ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 227–236, 2015.

6 Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Leslie Lamport. Adaptive register allocation with a linear number of registers. In *International Symposium on Distributed Computing*, DISC '13, pages 269–283, 2013.

7 Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Sergio Rajsbaum. Black art: Obstruction-free k-set agreement with |mwmr registers| < |proccesses|. In *1st International Conference on Networked Systems*, NETYS '13, pages 28–41, 2013.

8 Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Sergio Rajsbaum. Linear space bootstrap communication schemes. *Theoretical Computer Science*, 561:122–133, 2015.

9 Carole Delporte-Gallet, Hugues Fauconnier, Petr Kuznetsov, and Eric Ruppert. On the space complexity of set agreement? In *34th ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 271–280, 2015.

10 Panagiota Fatourou, Faith Ellen Fich, and Eric Ruppert. Time-space tradeoffs for implementations of snapshots. In *38th ACM Symposium on Theory of Computing*, STOC '06, pages 169–178, 2006.

11 Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 522–529, 2003.

12 Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for non-blocking implementations (preliminary version). In *15th ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 257–266, 1996.

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

14 F. P. Ramsey. On a problem of formal logic. *Proceedings of the London Mathematical Society*, 30:264–286, 1930.

15 Gadi Taubenfeld. Contention-sensitive data structures and algorithms. In *23rd International Conference on Distributed Computing*, DISC'09, pages 157–171, 2009.

16 Nayuta Yanagisawa. Wait-free solvability of colorless tasks in anonymous shared-memory model. In *18th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS '06, pages 415–429, 2016.

17 Leqi Zhu. A tight space bound for consensus. In *48th ACM Symposium on Theory of Computing*, STOC '16, pages 345–350, 2016.

A Proofs for Algorithm 1 (k-lock-free SWMR implementation)

▶ **Lemma 2.** *Let, at some point of a run of the algorithm, value (v, id, c) be present in some register r and such that no process is poised to execute an update on r (i.e., no process is between taking the snapshot of MEM (line 9) and the update of r (line 12)), then at all subsequent times $(v, id, c) \in r$, i.e., the value is present in the set of values stored in r.*

Proof. Suppose that at time τ, a register R contains (v, id, c) and no process is poised to execute an update on R. Suppose, by contradiction, that R does not contain it at some time $\tau' > \tau$. Let τ_{min}, $\tau_{min} > \tau$, be the smallest time such that (v, id, c) is not in R. Therefore, a write must have been performed on R, by some process q, at time τ_{min} with a view which does not contain (v, id, c). Such a write can only be performed at line 12, with a view including the last snapshot of MEM performed by q at line 9. Process q must have performed this snapshot operation on R at some $\tau_R < \tau$ as (v, id, c) is present in R between times τ and τ_{min} and as $\tau_R < \tau_{min}$. Thus q is poised to write on R at time τ — a contradiction. ◀

▶ **Lemma 3.** *If process p returns from a Write operation $(v, id(p), c)$ at time τ, then for any time $\tau' \geq \tau$ there is a register containing $(v, id(p), c)$.*

Proof. Before returning from its Write operation, p takes a snapshot of MEM at some time τ_S, $\tau_S < \tau$ (line 9), which returns a view of the memory in which at least n registers contain the triplet $(v, id(p), c)$. As p is taking a snapshot at line 9 at time τ_S, at most $n-1$ processes can be poised to perform an update on some register at time τ_S. As a process can be poised to perform an update on at most one register at a time, there can be at most $n-1$ distinct registers covered at time τ_S. Therefore, at time τ_S, there is at least one uncovered register containing $(v, id(p), c)$, let us call it r. By Lemma 2, $(v, id(p), c)$ will be present in r at any time $\tau' > \tau_S$, and thus at any time $\tau' > \tau$ as $\tau > \tau_S$. ◀

▶ **Lemma 5.** *Write operations in Algorithm 1 satisfy 1-lock-freedom.*

Proof. Suppose, by way of contradiction, that Write operations do not satisfy 1-lock-freedom. Then, eventually, all n first registers are infinitely often updated only by correct processes unsuccessfully trying to complete a Write operation. Thus, eventually each of the n first registers contain the value from one of these incomplete Write operations. As there are at most $n-1$ covered registers when a snapshot is taken, one of these value is eventually permanently present in some register (Lemma 2). This value is then eventually contained in the local view of every correct process, and thus, will eventually be present in every update of all the n first registers. The process with this Write value must therefore eventually pass the test on line 15 and, thus, complete its Write operation — a contradiction. ◀

▶ **Lemma 6.** *If a process q performing infinitely many operations sees $(v, id(p), c)$, and if p is correct, then p eventually completes its c^{th} Write operation.*

Proof. By Lemma 3, if process q returns from a Write operation with value $(v, id(q), c')$ at time τ, then for any time $\tau' \geq \tau$ there is a register containing $(v, id(q), c')$. But note that $(v, id(q), c')$ is written to a register only associated with q's local view. Thus, as q completes an infinite number of Write operations, each local view of q will eventually be forever present in some register. It will then eventually be observed in every snapshot taken by correct processes, and, therefore, included in their local view. This implies that it will eventually be present in every register written infinitely often, in particular in the first n registers. Process p will then eventually pass the test at line 15 and complete its corresponding c^{th} Write operation. ◀