Anonymous Obstruction-Free (n, k)-Set Agreement with $n - k + 1$ Atomic Read/Write Registers[*]

Zohir Bouzid[1], Michel Raynal[2], and Pierre Sutra[3]

1 IRISA, Université de Rennes, Rennes, France
2 IRISA, Université de Rennes, Rennes, France; and Institut Universitaire de France, Paris, France
3 University of Neuchâtel, Neuchatel, Switzerland; and Télécom SudParis, CNRS, Université Paris-Saclay, Paris, France

Abstract

The k-set agreement problem is a generalization of the consensus problem. Namely, assuming that each process proposes a value, every non-faulty process should decide one of the proposed values, and no more than k different values should be decided. This is a hard problem in the sense that we cannot solve it in an asynchronous system, as soon as k or more processes may crash. One way to sidestep this impossibility result consists in weakening the termination property, requiring that a process must decide a value only if it executes alone during a long enough period of time. This is the well-known obstruction-freedom progress condition.

Consider a system of n *anonymous asynchronous* processes that communicate through atomic *read/write registers*, and such that *any number of them may crash*. In this paper, we address and solve the challenging open problem of designing an obstruction-free k-set agreement algorithm using only $(n-k+1)$ atomic registers. From a shared memory cost point of view, our algorithm is the best algorithm known so far, thereby establishing a new upper bound on the number of registers needed to solve the problem, and in comparison to the previous upper bound, its gain is $(n-k)$ registers. We then extend this algorithm into a space-optimal solution for the repeated version of k-set agreement, and an x-obstruction-free solution that employs $(n - k + x)$ atomic registers (with $1 \leq x \leq k < n$).

1998 ACM Subject Classification D.1.3 [Software] Concurrent Programming, F.2 [Theory of Computation] Analysis of Algorithms and Problem Complexity, F.2.2 [Theory of Computation] Nonnumerical Algorithms and Problems

Keywords and phrases Anonymous processes, Asynchronous system, Atomic read/write register, Consensus, Fault-tolerance, k-Set agreement, Obstruction-freedom, Upper bound

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.18

1 Introduction

Due to failures, concurrent processes have to deal not only with finite asynchrony, i.e., finite but arbitrary process speed, but also with infinite asynchrony. In this context, mutex-based

[*] This work has been partially supported by the French ANR project DISPLEXITY devoted to computability and complexity in distributed computing, the Franco-German ANR project DISCMAT devoted to connections between mathematics and distributed computing, as well as the European Commission's Seventh Framework Program (FP7) under grant agreement No. 318809 (LEADS).

synchronization becomes useless, and pioneering works in *fault-tolerant* distributed computing, e.g., [21, 25], have instead promoted the design of concurrent algorithms [19, 26, 29].

A first challenge: multi-writer registers. When processes communicate with *Single-Writer Multi-Reader* (SWMR) atomic registers, a concurrent algorithm usually associates each process with a register. In the case where processes communicate with *Multi-Writer Multi-Reader* (MWMR) atomic registers, as any process can write any register, the previous association is no longer granted for free. To still benefit from existing SWMR registers-based solutions, a classical reduction consists in emulating SWMR registers on top of MWMR registers. In a system of n processes, it is shown [7, 9] that $(2n-1)$ MWMR atomic registers are needed to wait-free [16] simulate one SWMR atomic register, and that n MWMR atomic registers are needed if the simulation is required to be only non-blocking [20].[1] As a consequence, the simulation approach becomes irrelevant if the system provides less than n MWMR registers. In this context, the present paper focuses on what we name *genuine* concurrent algorithms, where "genuine" means "without simulating SWMR registers on top of MWMR registers". As underlined in [8], the design of genuine algorithms based on MWMR registers is still in its infancy, and sometimes resembles "black art" in the sense that the underlying intuition is difficult to grasp and formulate.

A second challenge: anonymity. Some algorithms based on MWMR registers, e.g., [26], require processes to write control values that include their identities. On the contrary, in an *anonymous* system, processes have no identity, the same code, and the same initialization of their local variables. Hence, they are in a strong sense identical. In such a context, the core question that interests us is the following: "Is it possible to solve a given problem with MWMR registers and anonymous processes, and if the answer is "yes", how many registers do we need ?"

Consensus and k-set agreement. We focus on the k-set agreement problem in a system of n processes. This problem introduced in [6], and denoted (n,k)-set agreement in the following, is a generalization of consensus, which corresponds to the case where $k=1$. Assuming that each participating process proposes a value, every non-faulty process must decide a value (termination), which was proposed by some process (validity), and at most k different values are decided (agreement).

Impossibility results and the case of obstruction-freedom. Designing a deterministic wait-free consensus in an asynchronous system prone to even a single crash failure is not possible [12, 23]. If now k or more processes may crash, there is no deterministic wait-free read/write solution to (n,k)-set agreement [4, 18, 27]. As we are interested in the computing power of *pure read/write* asynchronous systems, we neither want to enrich the underlying system with additional power (e.g., synchrony assumptions, random numbers, or failure detectors), nor impose constraints on the input vector collectively proposed by the processes. To sidestep the above impossibility result, we thus consider a progress property weaker than wait-freedom, namely *obstruction-freedom* [17]. For (n,k)-set agreement, this property states that a process decides a value only if it executes solo during a "long enough" period of time without

[1] "Wait-free" means that any read or write invocation on the SWMR register that is built must terminate if the invoking process does not crash [16]. "Non-blocking" means that at least one process that does not crash returns from all its read and write invocations [20].

interruption. The notion of x-obstruction-freedom [30] generalizes this idea to any group of at most x processes.

Contributions of the paper. This paper details a *genuine obstruction-free* algorithm solving the (n, k)-set agreement problem in an *asynchronous anonymous read/write* system where any number of processes may crash. Our algorithm makes use of $(n - k + 1)$ MWMR registers, i.e., exactly n registers for consensus. For (n, k)-set agreement, the best lower bound known so far [10] is $\Omega(\sqrt{\frac{n}{k} - 2})$, while the best obstruction-free (n, k)-set agreement algorithm requires $2(n - k) + 1$ MWMR registers [8, 10]. As a consequence, our algorithm provides a gain of $(n - k)$ MWMR registers. In the case of consensus, Gelashvili [14] proved recently that $n/20$ registers are necessary, and Zhu [31] improved this bound to $n - 1$. Hence, our algorithm is up to an additive factor of 1 close to the best known lower bound.

In the *repeated* version of the (n, k)-set agreement problem, processes participate in a sequence of (n, k)-set agreement instances. It was recently proved [10] that $(n - k + 1)$ atomic registers are necessary to solve repeated (n, k)-set agreement. This paper shows that a simple modification of our base construction solves *repeated* (n, k)-set agreement without additional atomic registers. The resulting algorithm is thus optimal, closing the gap on previous proposed upper bounds for this problem.

Our construction is round-based, following the pattern “snapshot; local computation; write”, where the snapshot and write operations occur on the $(n-k+1)$ MWMR registers. This pattern is reminiscent of the one named “look; compute; move” found in robot algorithms [28]. Interestingly, processes do not maintain any local information between successive rounds. In this sense, our algorithm is *locally memoryless*. Each register contains a quadruplet consisting of a round number, two control bits, and a proposed value. The algorithm exploits a partial order over the quadruplets. The way a process computes a new quadruplet is the key of our algorithm. The variation for repeated (n, k)-set agreement employs sixuplets.

Roadmap. Section 2 presents the computing model and definitions used in this paper. Section 3 depicts a base anonymous obstruction-free algorithm solving consensus; this algorithm captures the essence of our solution. We prove its correctness in Section 4. Section 5 extends our algorithm to solve (n, k)-set agreement, We address the case where (n, k)-set agreement is used repeatedly in Section 6. Section 7 considers the x-obstruction-freedom progress condition, and presents a solution using $(n - k + x)$ registers. We conclude in Section 8. Due to space constraints, we defer some details to our companion technical report [5].

2 Context & Problem Definition

2.1 Computing Model

We assume a distributed system of n asynchronous processes $\{p_1, \ldots, p_n\}$. When considering a process p_i, we name integer i its *index*. Indexes are used to ease the exposition from an external observer point of view. Processes do not have identities and execute the very same code. We assume that they know the value n.

Let $\mathbb{T}$ denote the increasing sequence of time instants (observable only from an external point of view). At each instant, a unique process is activated to execute a step. A *step* consists in a read or a write to a register (access to the shared memory) possibly followed by a finite number of internal operations (on local variables).

Up to $(n-1)$ processes may crash. A crash is an unexpected halting. After it has crashed (if it ever does), a process remains crashed forever. From a terminology point of view, and given an execution, a *faulty* process is a process that crashes, and a *correct* process is a process that does not crash.[2]

In addition to processes, the computing model includes a communication medium made up of m multi-writer/multi-reader (MWMR) registers.[3] Registers are encapsulated in an array denoted $REG[1..m]$. The registers are *atomic*. This means that read and write operations appear as if they have been executed sequentially, and this sequence (a) respects the real-time order of non-concurrent operations, and (b) is such that each read returns the value written by the closest preceding write operation [22]. When considering some concurrent object defined from a sequential specification, atomicity is named *linearizability* [20].

From atomic registers to a snapshot object. At the upper layer (where consensus and (n, k)-set agreement are solved), we use the array $REG[1..m]$ to construct a snapshot object [1]. This object, denoted REG hereafter, provides processes with the operations write() and snapshot(). When a process invokes REG.write(x, v), it deposits the value v in $REG[x]$. When it invokes REG.snapshot() it obtains the content of the whole array. The snapshot object is linearizable, i.e., every invocation of REG.snapshot() appears as instantaneous. For the REG object, a linearization is a sequence of write and snapshot operations.

An anonymous non-blocking (hence obstruction-free) implementation of a snapshot object is described in [15]. This implementation does not require additional atomic registers. In the following, we consider that the snapshot object REG is implemented using this algorithm.

2.2 Obstruction-free consensus and obstruction-free (n, k)-set agreement

Obstruction-free consensus. An obstruction-free consensus object is a one-shot object that provides each process with a single operation denoted propose(). This operation takes a value as input parameter and returns a value.

"*One-shot*" means that a process invokes propose() at most once. When a process invokes propose(v), we say that it "proposes v". When the invocation of propose() returns value v', we say that the invoking process "decides v'". A process executes "solo" when it keeps on executing while the other processes have stopped their execution (at any point of their algorithm). The obstruction-free consensus problem is defined by the following properties (that is, to be correct, any obstruction-free algorithm must satisfy such properties).

- Validity. If a process decides a value v', this value was proposed by a process.
- Agreement. No two processes decide different values.
- OB-termination. If there is a time after which a process executes solo, it decides a value.
- SV-termination[4]. If a single value is proposed, all correct processes decide.

Validity relates outputs to inputs. Agreement relates the outputs. Termination states the conditions under which a correct process must decide. There are two cases. The first is

[2] No process knows if it is correct or faulty. This is because, before crashing, a faulty process behaves as a correct one.
[3] As pointed out in the introduction, we recall that anonymity prevents processes from using single-writer/multi-reader registers.
[4] This termination property, which relates termination to the input values, is not part of the classical definition of the obstruction-free consensus problem. It is an additional requirement which demands termination under specific circumstances that are independent of the concurrency pattern.

```
function sup(T) is
(S1) let $\langle r, \ell evel, -, v\rangle$ be $\mathsf{max}(T)$;
(S2) let $vals(T)$ be $\{w \mid \exists\langle r, -, -, w\rangle \in T\}$;
(S3) let $conf\ell ict1(T)$ be $\exists\ \langle r, -, \mathtt{true}, -\rangle \in T$;
(S4) let $conf\ell ict2(T)$ be $|vals(T)| > 1$;
(S5) let $conf\ell ict(T)$ be $conf\ell ict1(T) \vee conf\ell ict2(T)$;
(S6) return$\big(\langle r, \ell evel, conf\ell ict(T), v\rangle\big)$.
```

Figure 1 The function sup().

related to obstruction-freedom. The second one is independent of the concurrency and failure pattern; it is related to the input value pattern.

Obstruction-free (n, k)-set agreement. An obstruction-free (n, k)-set agreement object is a one-shot object which has the same validity, OB-termination, and SV-termination properties as consensus, and for which we replace the agreement property with:

- Agreement. At most k different values are decided.

As for consensus, SV-termination property is a new property strengthening the classical definition of k-set agreement [6].

In what follows, we describe first an obstruction-free anonymous algorithm that solves the consensus problem, then we extend it to address (n, k)-set agreement.

3 Obstruction-free Anonymous Consensus Algorithm

Our consensus algorithm is detailed in Figure 2. As indicated in the Introduction, its essence is captured by the quadruplets that can be written in the MWMR atomic registers.

Shared memory. The shared memory is made up of a snapshot object REG, composed of $m = n$ MWMR atomic registers. Each of them contains a quadruplet initialized to $\langle 0, \mathtt{down}, \mathtt{false}, \perp\rangle$. The meaning of these fields is the following.

- The first field, denoted rd, is a round number.
- The second field, denoted $\ell v\ell$ (level), has a value in $\{\mathtt{up}, \mathtt{down}\}$, where $\mathtt{up} > \mathtt{down}$.
- The third field, denoted $cf\ell$ (conflict), is a Boolean (initially equals to $\mathtt{false}$). We assume $\mathtt{true} > \mathtt{false}$.
- The last field, denoted $va\ell$, is initialized to $\perp$, and then contains always a proposed value. It is assumed that the set of proposed values is totally ordered, and that $\perp$ is smaller than any proposed value.

When considering the lexicographical ordering, it is easy to see that all the possible quadruplets $\langle rd, \ell v\ell, cf\ell, va\ell\rangle$ form a totally ordered set. This total order, and its reflexive closure, are denoted "$<$" and "$\leq$", respectively.

Notion of conflict and the function sup(). The function sup(), defined in Figure 1, plays a central role in our algorithm. It takes a non-empty set of quadruplets T as input parameter, and returns a quadruplet, which is the supremum of T, defined as follows.

Let $\langle r, \ell evel, -, v\rangle$ be the maximal element of T according to the lexicographical ordering (line S1), and $vals(T)$ be the values in the quadruplets of T associated with the maximal round number r (line S2). The set T is *conflicting* if one of the two following cases occurs (line S5).

```
operation propose($v_i$) is
(01) repeat forever
(02)     $view \leftarrow REG.\mathsf{snapshot}()$;
(03)     case $(\exists r > 0, val : \forall x : view[x] = \langle r, \mathtt{up}, \mathtt{false}, val\rangle)$ then
                 $\mathsf{return}(val)$
(04)          $(\exists r > 0, val : \forall x : view[x] = \langle r, \mathtt{down}, \mathtt{false}, val\rangle)$ then
                 $REG.\mathsf{write}(1, \langle r+1, \mathtt{up}, \mathtt{false}, val\rangle)$
(05)          $(\exists r > 0, val, level : \forall x : view[x] = \langle r, level, \mathtt{true}, val\rangle)$ then
                 $REG.\mathsf{write}(1, \langle r+1, \mathtt{down}, \mathtt{false}, val\rangle)$;
(06)          otherwise let $\langle r, level, cfl, val\rangle$
                          $\leftarrow \mathsf{sup}(view[1], \cdots, view[n], \langle 1, \mathtt{down}, \mathtt{false}, v_i\rangle)$;
(07)                $x \leftarrow$ smallest index such that $view[x] \neq \langle r, level, cfl, val\rangle$;
(08)                $REG.\mathsf{write}(x, \langle r, level, cfl, val\rangle)$
(09)     end case
(10) end repeat.
```

Figure 2 Anonymous obstruction-free Consensus.

- There is a quadruplet $X = \langle r, -, \mathtt{true}, -\rangle$ in T (line S3). In this case, there is a quadruplet $X \in T$ whose round number is the highest ($X.rd = r$), and whose conflict field $X.\ell v\ell = \mathtt{true}$. We then say that the conflict is "inherited".
- There are at least two quadruplets X and Y in T, that have the highest round number in T (i.e., $X.rd = Y.rd = r$), and that contain two different values (i.e., $X.val \neq Y.val$) (lines S2 and S4). In such a case, we say that the conflict is "discovered".

Function $\mathsf{sup}(T)$ first checks whether T is conflicting (lines S2–S5). Then, it returns at line S6 the quadruplet $\langle r, level, conflict(T), v\rangle$, where $conflict(T)$ indicates if the input set T is conflicting (line S5). Let us notice that, since $\mathtt{true} > \mathtt{false}$, the quadruplet returned by $\mathsf{sup}(T)$ is always greater than, or equal to, the greatest element in T, i.e., $\mathsf{sup}(T) \geq \mathsf{max}(T)$.

The algorithm. Our base construction is pretty simple, and consists in an appropriate management of the snapshot object REG, so that the n quadruplets it contains (a) never allow validity or agreement to be violated, and (b) eventually allow termination under good circumstances (which occur when obstruction-freedom is satisfied or when a single value is proposed).

In Figure 2, when a process p_i invokes $\mathsf{proposes}(v_i)$, it enters a loop that it will exit at line 03 (provided it terminates) with the statement $\mathsf{return}(val)$, where val is the decided value. After entering the loop, a process issues a snapshot and assigns the returned array to its local variable $view[1..n]$ (line 02). Then, there are two main cases according to the value stored in $view$.

- Case 1 (lines 03–05). All entries of $view_i$ contain the same quadruplet $\langle r, level, conflict, val\rangle$, and $r > 0$. Then, there are three sub-cases to consider.
  - Case 1.1. If the level is $\mathtt{up}$ and the conflict is $\mathtt{false}$, the invoking process decides the value val (line 03).
  - Case 1.2. If the level is $\mathtt{down}$ and the conflict field is $\mathtt{false}$, process p_i enters the next round by writing $\langle r+1, \mathtt{up}, \mathtt{false}, val\rangle$ in the first entry of REG (line 04).
  - Case 1.3. If there is a conflict, p_i enters the next round by writing $\langle r+1, \mathtt{down}, \mathtt{false}, val\rangle$ in the first entry of REG (line 05).

- Case 2 (lines 06–08). Not all the entries of $view_i$ are equal, or one of them contains a tuple $\langle 0, -, -, -\rangle$. In such a case, p_i first calls $\texttt{sup}(view[1], \cdots, view[n], \langle 1, \texttt{down}, \texttt{false}, v_i\rangle)$ (line 06), which returns a quadruplet X greater than all the input quadruplets, or equal to the greatest of them. As we have seen previously, this quadruplet X may inherit or discover a conflict. Moreover, as $\langle 1, \texttt{down}, \texttt{false}, v_i\rangle$ is an input parameter of $\texttt{sup}()$, $X.val$ cannot equal $\perp$. As none of the predicates at lines 03–05 is satisfied, at least one entry of $view[1..n]$ is different than X. Process p_i writes then X into $REG[x]$, where, from its point of view, x is the first entry of REG whose content differs from X (lines 07–08).

The underlying operational intuition. To understand the intuition that underlies our algorithm, let us first consider the very simple case where a single process p_i executes the algorithm. From its first invocation of REG.snapshot() (line 02), it obtains a view $view$ in which all the elements are equal to $\langle 0, \texttt{down}, \texttt{false}, \perp\rangle$. Hence, p_i executes line 06, where the invocation of $\texttt{sup}()$ returns the quadruplet $\langle 1, \texttt{down}, \texttt{false}, v_i\rangle$, that is written into $REG[1]$ at line 08. Then, during the second round, p_i computes with the help of function $\texttt{sup}()$ again the quadruplet $\langle 1, \texttt{down}, \texttt{false}, v_i\rangle$, and p_i writes it into $REG[2]$; etc., until p_i writes $\langle 1, \texttt{down}, \texttt{false}, v_i\rangle$ in all the atomic registers of $REG[1..n]$. When this occurs, p_i obtains at line 02 a view where all the elements equal $\langle 1, \texttt{down}, \texttt{false}, v_i\rangle$. It consequently executes line 04 and writes $\langle 2, \texttt{up}, \texttt{false}, v_i\rangle$ in $REG[1]$. Then, during the following rounds, process p_i writes $\langle 2, \texttt{up}, \texttt{false}, v_i\rangle$ in the other registers of REG (line 08). When this is done, p_i obtains a snapshot containing solely $\langle 2, \texttt{up}, \texttt{false}, v_i\rangle$, and when this occurs, p_i executes line 03 where it decides the value v_i.

Let us now consider the case where, while p_i is executing, another process p_j invokes propose(v_j) with $v_j = v_i$. It is easy to see that, in such a case, p_i and p_j collaborate to fill in REG with the same quadruplet $\langle 2, \texttt{up}, \texttt{false}, v_i\rangle$. If $v_j \neq v_i$, depending on the concurrency pattern, a conflict may occur. For instance, it occurs if REG contains both $\langle 1, \texttt{down}, \texttt{false}, v_i\rangle$ and $\langle 1, \texttt{down}, \texttt{false}, v_j\rangle$. If a conflict appears, it will be propagated from round to round, until a process executes alone a higher round.

▶ Remark 1. We first notice that no process needs to memorize in its local memory the values that it will use during the next rounds. Not only processes are anonymous, but their code is also memoryless (no persistent variables). The snapshot object REG constitutes the whole memory of the system. Hence, as defined in the Introduction, the algorithm is locally memoryless. In this sense, and from a locality point of view, it has a “functional” flavor.

▶ Remark 2. Let us consider the n-bounded concurrency model [2, 24]. This model is made up of an arbitrary number of processes, but, at any time, there are at most n processes executing steps. This allows processes to leave the system and other processes to join it as long as the concurrency degree does not exceed n.

The previous algorithm works without modification in such a model. A proposed value is now a value proposed by any of the N processes that participate in the algorithm. Hence, if $N > n$, the number of proposed values can be greater than the upper bound n on the concurrency degree. This versatility dimension of our algorithm is a direct consequence of the previous “locally memoryless” property.

4 Proof of the Algorithm

In this section, we present the correctness proof of our obstruction-free anonymous consensus algorithm. After a few definitions provided in Section 4.1, Section 4.2 shows that a relation “$\sqsupseteq$” defined over the quadruplets is a partial order. This relation is central to prove the key properties of our algorithm. Such properties are established in Sections 4.3 and 4.4. Then, based on these properties, Section 4.5 shows that our algorithm is correct. Notice that, due to space restrictions, we state some lemmata without their corresponding proofs. The interested reader will find them in our companion technical report [5].

4.1 Definitions and notations

Let $\mathcal{E}$ be a set of quadruplets that can be written in *REG*. Given $X \in \mathcal{E}$, its four fields are denoted $X.rd, X.\ell v\ell, X.cf\ell$ and $X.va\ell$, respectively. Relations $>$ and $\geq$ refer to the lexicographical ordering over $\mathcal{E}$. Moreover, where appropriate, we consider the array $view[1..n]$ as the set $\{view[1], \cdots, view[n]\}$.

▶ **Definition 3.** Let $X, Y \in \mathcal{E}$.

$$X \sqsupset Y \overset{def}{=} (X > Y) \wedge [(X.rd > Y.rd) \vee (X.cf\ell) \vee (\neg Y.cf\ell \wedge X.va\ell = Y.va\ell)].$$

At the operational level the algorithm ensures that the quadruplets it generates are totally ordered by the relation $>$. Differently, the relation $\sqsupset$ (which is a partial order on these quadruplets, see Section 4.2) captures the relevant part of this total order, and is consequently the key cornerstone on which the proof of our algorithm relies.

When $X \sqsupset Y$ holds, we say that "*X strictly dominates Y*". Similarly, *X dominates Y*, written $X \sqsupseteq Y$, when $(X \sqsupset Y)$ or $(X = Y)$ holds. Relations $\sqsubset$ and $\sqsubseteq$ are defined in the natural way.

▶ **Definition 4.** Given a set of quadruplets T, we shall say that T is *homogeneous* when it contains a single element, say X. We then write it "T is $\mathcal{H}(X)$".

▶ Notation 1. The value, at time τ, of the local variable α of a process p_i is denoted α_i^τ. Similarly the value of an atomic register $REG[x]$ at time τ is denoted $REG^\tau[x]$, and the value of REG at time τ is denoted REG^τ.

▶ Notation 2. We note $\mathcal{W}(x, X)$ the writing of a quadruplet X in the register $REG[x]$.

▶ **Definition 5.** We say "a process p_j *covers* $REG[x]$ at time τ" when its next non-local step after time τ is $\mathcal{W}(x, X)$, where X is the quadruplet which is written. In this case we also say "$\mathcal{W}(x, X)$ *covers* $REG[x]$ at time τ" or "$REG[x]$ *is covered* by $\mathcal{W}(x, X)$ at time τ".

Let us notice that if p_j covers $REG[x]$ at time τ, then τ necessarily lies between the last snapshot issued by p_j at line 02 and its planned write $\mathcal{W}(x, X)$ that will occur at line 04, 05, or 08.

4.2 The relation $\sqsupseteq$ is a partial order

▶ **Lemma 6.** $((X \sqsupset Y \sqsupset Z) \wedge (X.rd = Y.rd = Z.rd)) \Rightarrow (X.cf\ell \vee (\neg Z.cf\ell \wedge X.va\ell = Z.va\ell))$.

▶ **Lemma 7.** $\sqsupseteq$ *is a partial order.*

4.3 Extracting the relations $\sqsupset$ and $\sqsupseteq$ from the algorithm

The definition of sup() appears in Figure 1.

▶ **Lemma 8.** *Let T be a set of quadruplets. For every $X \in T$: $\mathsf{sup}(T) \sqsupseteq X$.*

▶ **Lemma 9.** *If p_i executes $\mathcal{W}(-, Y)$ at time τ, then for every $X \in view_i^\tau$: $Y \sqsupseteq X$.*

▶ **Lemma 10.** *Let us assume that no process is covering $REG[x]$ at time τ. For every write $\mathcal{W}(-,X)$ that (a) occurs after τ and (b) was not covering a register of REG at time τ, we have $X \sqsupseteq REG^{\tau}[x]$.*

Proof. The proof is by contradiction. Let p_i be the first process that executes a write $\mathcal{W}(-,X)$ contradicting the lemma. This means that $\mathcal{W}(-,X)$ is not covering a register of REG at time τ and $X \not\sqsupseteq REG^{\tau}[x]$. Let this write occur at time $\tau_2 > \tau$. Thus, all writes that take place between τ and τ_2 comply with the lemma. We derive a contradiction by showing that $X \sqsupseteq REG^{\tau}[x]$.

Let $\tau_1 < \tau_2$ be the linearization time of the last snapshot taken by p_i (line 02) before executing $\mathcal{W}(-,X)$. Since $\mathcal{W}(-,X)$ was not covering a register of REG at time τ, the snapshot preceding this write was necessarily taken after τ. That is, $\tau_1 > \tau$, and we have $\tau_2 > \tau_1 > \tau$.

According to Lemma 9, $X \sqsupseteq view_i^{\tau_2}[x]$. But since the snapshot returning $view_i^{\tau_2}$ is linearized at τ_1, it follows that $view_i^{\tau_2} = REG^{\tau_1}$. Therefore, we have $X \sqsupseteq REG^{\tau_1}[x]$ (assertion R).

In the following we show that $REG^{\tau_1}[x] \sqsupseteq REG^{\tau}[x]$. If $REG[x]$ was not updated between τ and τ_1, then $REG^{\tau_1}[x] = REG^{\tau}[x]$ and the claim follows. Otherwise, if $REG[x]$ was updated between τ and τ_1, the content of $REG^{\tau_1}[x]$, let it be Y, is the result of a write $\mathcal{W}(x,Y)$ that occurred between τ and τ_1 and that was not covering a register of REG at time τ (remember that no write is covering $REG[x]$ at time τ). We assumed above that τ_2 is the first time at which the lemma is contradicted. Hence the write $\mathcal{W}(x,Y)$, which occurs before τ_2, complies with the requirements of the lemma. It follows that $Y \sqsupseteq REG^{\tau}[x]$, and we consequently have $REG^{\tau_1}[x] \sqsupseteq REG^{\tau}[x]$.

But it was shown above (see assertion R) that $X \sqsupseteq REG^{\tau_1}[x]$. Hence, due to the transitivity of the relation $\sqsupseteq$ (Lemma 7), we obtain $X \sqsupseteq REG^{\tau}[x]$, a contradiction that concludes the proof of the lemma. ◀

▶ **Lemma 11.** *Let τ and $\tau' \geq \tau$ be two time instants. If $REG^{\tau'}$ is $\mathcal{H}(Y)$, then there exists $X \in REG^{\tau}$ such that $Y \sqsupseteq X$.*

The following two lemmata are corollaries of Lemma 11.

▶ **Lemma 12.** *If REG^{τ} is $\mathcal{H}(X)$, $REG^{\tau'}$ is $\mathcal{H}(Y)$, and $\tau' \geq \tau$, then $Y \sqsupseteq X$.*

▶ **Lemma 13.** *If REG^{τ} is $H(X)$, $REG^{\tau'}$ is $\mathcal{H}(Y)$, $\tau' \geq \tau$, $(Y.rd = X.rd)$ and $(\neg Y.cf\ell)$ then $(Y.va\ell = X.va\ell)$.*

4.4 Exploiting homogeneous snapshots

▶ **Lemma 14.** $[(X \in REG^{\tau}) \wedge (X.\ell v\ell = \texttt{up})] \Rightarrow \big(\exists\ \tau' < \tau$*: $REG^{\tau'}$ is $\mathcal{H}(Z)$, where* $Z = \langle X.rd - 1, \texttt{down}, \texttt{false}, X.va\ell\rangle\big)$.

Proof. Let us first show that there is a process that writes the quadruplet X' into REG, with $X' = \langle X.rd, X.\ell v\ell, \texttt{false}, X.va\ell\rangle$. We have two cases depending on the value of $X.cf\ell$.

- If $X.cf\ell = \texttt{false}$, then let $X' = X$. Since $X.\ell v\ell = X'.\ell v\ell = \texttt{up}$, X was necessarily written into REG by some process (let us recall that the initial value of each register of REG is $\langle 0, \texttt{down}, \texttt{false}, \perp\rangle$).
- If $X.cf\ell = \texttt{true}$, let us consider the time τ_1 at which X was written for the first time into REG, say by p_i. Since $X.\ell v\ell = \texttt{up}$, both τ_1 and p_i are well defined. This write of X happens necessarily at line 08 (If it was at line 04 or 05, we would have $X.cf\ell = \texttt{false}$).

Therefore, X was computed at line 06 by the function $\mathsf{sup}()$. Namely we have $X = \mathsf{sup}(T)$, where the set T is equal to $\{view^{\tau}[1], \cdots, view^{\tau}[n], \langle 1, \mathtt{down}, \mathtt{false}, v_i\rangle\}$. Observe that $X \notin T$, otherwise X would not be written for the first time at τ_1. Let $X' = \mathsf{max}(T)$. Since $X \notin T$, it follows that $X \neq X'$. Due to line S6 of the function $\mathsf{sup}()$, X and X' differ only in their conflict field. Therefore, as $X.cf\ell = \mathtt{true}$, it follows that $X'.cf\ell = \mathtt{false}$. Finally, as $X'.\ell v\ell = \mathtt{up}$ and all registers of REG are initialized to $\langle 0, \mathtt{down}, \mathtt{false}, \bot\rangle$, it follows that X' was necessarily written into REG by some process.

In both cases, there exists a time at which a process writes $X' = \langle X.rd, X.\ell v\ell, \mathtt{false}, X.va\ell\rangle$ into REG. Let us consider the first process p_i that does so. This occurs at some time $\tau_2 < \tau$. As $X'.\ell v\ell = \mathtt{up}$, this write can occur only at line 04 or line 08.

We first show that this write occurs necessarily at line 04. Assume for contradiction that the write of X' into REG happens at line 08. In this case, the quadruplet X' was computed at line 06. Therefore, $X' = \mathsf{sup}(T)$ where where the set T is equal to $\{view^{\tau_2}[1], \cdots, view^{\tau_2}[n], \langle 1, \mathtt{down}, \mathtt{false}, v_i\rangle\}$. Observe that $\mathsf{sup}(T)$ and $\mathsf{max}(T)$ can differ only in their conflict field. As $\mathsf{sup}(T).cf\ell = X'.cf\ell = \mathtt{false}$, it follows that $X' = \mathsf{sup}(T) = \mathsf{max}(T)$. Consequently, $X' \in view^{\tau_2}$. That is, p_i is not the first process that writes X' in REG, contradiction. Therefore, the write necessarily happens at line 04.

From the precondition at line 04, $view^{\tau_2}$ is $\mathcal{H}(\langle X'.rd - 1, \mathtt{down}, \mathtt{false}, X'.va\ell\rangle)$, and the lemma holds. ◀

▶ **Lemma 15.** *$[(REG^{\tau}$ is $\mathcal{H}(X)) \wedge (X.\ell v\ell = \mathtt{up}) \wedge (\neg X.cf\ell) \wedge (REG^{\tau'}$ is $\mathcal{H}(Y)) \wedge (Y.rd \geq X.rd)] \Rightarrow (Y.va\ell = X.va\ell)$.*

4.5 Proof of the algorithm: using the previous lemmata

▶ **Lemma 16.** *No two processes decide different values.*

Proof. Let r be the smallest round in which a process decides, p_i and $va\ell$ being the deciding process and the decided value, respectively. There is a time τ at which $view_i^{\tau}$ is $\mathcal{H}(\langle r, \mathtt{up}, \mathtt{false}, va\ell\rangle)$. Due to Lemma 15, every homogeneous snapshot starting from round r is necessarily associated with the value $va\ell$. Therefore, only this value can be decided in any round higher than r. Since r was assumed to be the smallest round in which a decision occurs, the consensus agreement property follows. ◀

▶ **Lemma 17.** *For every quadruplet X that is written in REG, $X.va\ell$ is a value proposed by some process.*

▶ **Lemma 18.** *A decided value is a proposed value.*

▶ **Lemma 19.** *Let T be a set of quadruplets. For every $T' \subseteq T$: $\mathsf{sup}(T' \cup \{\mathsf{sup}(T)\}) = \mathsf{sup}(T)$.*

▶ **Lemma 20.** *If there is a time after which a process executes solo, it decides a value.*

▶ **Lemma 21.** *If a single value is proposed, all correct processes decide.*

▶ **Theorem 22.** *The algorithm described in Figure 2 solves the obstruction-free consensus problem (as defined in Section 2.2).*

Proof. The proof follows directly from the Lemma 16 (Agreement), Lemma 18 (Validity), Lemma 20 (OB-Termination), and Lemma 21 (SV-Termination). ◀

5 From Consensus to (n,k)-Set Agreement

The algorithm. Our obstruction-free (n,k)-set agreement algorithm is the same as the one of Figure 2, except that now there are only $m = n - k + 1$ MWMR atomic registers instead of $m = n$. Hence REG is now $REG[1..(n-k+1)]$.

Its correctness. The arguments for the validity and liveness properties are the same as the ones of the consensus algorithm since they do not depend on the size of REG.

As far as the k-set agreement property is concerned (no more than k different values are decided), we have to show that $(n-k+1)$ registers are sufficient. To this end, let us consider the $(k-1)$ first decided values, where the notion "first" is defined with respect to the linearization time of the snapshot invocation (line 02) that immediately precedes the invocation of the corresponding deciding statement (return() at line 03). Let τ be the time just after the linearization of these $(k-1)$ "deciding" snapshots. Starting from τ, at most $(n-(k-1)) = (n-k+1)$ processes access the array REG, which is made up of exactly $(n-k+1)$ registers. Hence, after time τ, these $(n-k+1)$ processes execute the consensus algorithm of Figure 2, where $(n-k+1)$ replaces n, and consequently at most one new value is decided. Therefore, at most k values are decided by the n processes.

6 From One-shot to Repeated (n,k)-Set Agreement

6.1 The repeated (n,k)-set agreement problem

In the repeated (n,k)-set agreement problem, processes executes a sequence of (n,k)-set agreement instances. Hence, a process p_i invokes sequentially the operation propose$(1, v_i)$, then propose$(2, v_i)$, etc., where $sn_i = 1,\ 2,\ \dots$ is the sequence number of its current instance, and v_i is the value it proposes to this instance.

It would be possible to associate a specific instance of the base algorithm described in Figure 2 with each sequence number, but this would require $(n-k+1)$ atomic read/write registers per instance. The next section shows that we can solve the repeated problem with only $(n-k+1)$ atomic registers. According to the complexity results of [10], it follows that this algorithm is optimal in the number of atomic registers it uses, which consequently closes the space complexity discussion regarding repeated (n,k)-set agreement.

6.2 Adapting the algorithm

From quadruplets to sixuplets. Instead of a quadruplet, an atomic read/write register is now a sixuplet $X = \langle sn, rd, \ell v\ell, cf\ell, va\ell, dcd\rangle$. The four fields $X.rd$, $X.\ell v\ell$, $X.cf\ell$, $X.va\ell$ are the same as before. The additional field $X.sn$ contains a sequence number, while the other additional field $X.dcd$ is an initially empty list. From a notational point of view, the jth element of this list is denoted $X.dcd[j]$; it contains a value decided by the jth instance of the repeated (n,k)-set agreement.

The total order on sixuplets "$>$" is the classical lexicographical order defined on the first five fields, while relation "$\sqsupset$" is now defined as follows:

$$X \sqsupset Y \overset{def}{=} (X > Y) \wedge [(X.sn > Y.sn) \vee (X.rd > Y.rd) \vee (X.cf\ell) \vee (\neg Y.cf\ell \wedge X.va\ell = Y.va\ell)].$$

Local variables. Each process p_i now manages two local variables whose scope is the whole repeated (n,k)-set agreement problem.

```
operation propose($sn_i, v_i$) is
(01) repeat forever
(02)   $view \leftarrow REG.\mathsf{snapshot}()$;
(03)   case $(\exists r > 0, val : \forall x : view[x] = \langle sn_i, r, \mathtt{up}, \mathtt{false}, val, - \rangle)$ then
                 $dcd_i[sn_i] \leftarrow val$; $\mathsf{return}(val)$
(04)        $(\exists r > 0, val : \forall x : view[x] = \langle sn_i, r, \mathtt{down}, \mathtt{false}, val, - \rangle)$ then
                 $REG.\mathsf{write}(1, \langle sn_i, r+1, \mathtt{up}, \mathtt{false}, val, dcd_i \rangle)$
(05)        $(\exists r > 0, val, level : \forall x : view[x] = \langle sn_i, r, level, \mathtt{true}, val, - \rangle)$ then
                 $REG.\mathsf{write}(1, \langle sn_i, r+1, \mathtt{down}, \mathtt{false}, val, dcd_i \rangle)$
(06)        otherwise let $\langle inst, r, level, conflict, val, dec \rangle$
                          $\leftarrow \mathsf{sup}(view[1], \cdots, view[n], \langle sn_i, 1, \mathtt{down}, \mathtt{false}, v_i, dcd_i \rangle)$;
(07)                 if $(inst > sn_i)$ then $dcd_i[sn_i] \leftarrow dec[sn_i]$; return $dcd_i[sn_i]$ end if
(08)                 $x \leftarrow$ smallest index such that $view[x] = \min(view[1], \cdots, view[n])$;
(09)                 $REG.\mathsf{write}(x, \langle inst, r, level, conflict, val, dec \rangle)$
(10)   end case
(11) end repeat.
```

Figure 3 Repeated obstruction-free Consensus.

- The variable sn_i, initialized to 0, is used by p_i to generate its sequence numbers. It is assumed that p_i increments sn_i before invoking propose(sn_i, v_i).
- The local list dcd_i is used by p_i to store the value it has decided during the previous instances of the (n, k)-set agreement. Hence, $dcd_i[j]$ contains the value decided by p_i during the jth instance.

The algorithm. The algorithm executed by a process p_i is described in Figure 3. Parts that are new with respect to the base algorithm appear in blue in Figure 2. We detail the internals of our construction below.

- Line 03. When all the entries of the view obtained by p_i contain only sixuplets whose first five fields are equal, p_i decide the value val. But before returning val, p_i writes val in $dcd_i[sn_i]$. The idea is that, when p_i will execute the next (n, k)-set agreement instance (whose sequence number will be $sn_i + 1$), it will be able to help processes, whose current sequence number sn' are smaller than sn_i.
- Line 04. In this case, p_i obtains a view where the first five entries are equal to $\langle sn_i, r, \mathtt{down}, \mathtt{false}, val \rangle$. It then writes in $REG[1]$ the value $\langle sn_i, r, \mathtt{down}, \mathtt{false}, val, dcd_i \rangle$.
- Line 05. Similar to the previous case, except that a conflict now appears in the view of p_i.
- Lines 06-10. Process p_i computes the supremum of the snapshot value $view$ obtained at line 03 plus the sixuplet $\langle sn_i, 1, \mathtt{down}, \mathtt{false}, val, dcd_i \rangle$. There are two cases to consider.
  - If the sequence number of this supremum $inst$ is greater than sn_i (line 07), p_i can benefit from the list of values already decided in (n, k)-set agreement instances whose sequence number is greater than sn_i. This help is obtained from $dec[sn_i]$. Consequently, similarly to line 03, p_i writes this value in $dcd_i[sn_i]$ and decides it.
  - If now $inst$ equals sn_i, process p_i executes the same operations as in our base algorithm (lines 08–09).

It follows from the algorithm depicted in Figure 3 that solving repeated (n, k)-set agreement in an anonymous system does not require more atomic read/write registers than the base non-repeated version. The only additional cost lies in the size of the atomic registers which contain two supplementary unbounded fields. As pointed out in the introduction, the lower bound established in [10] induces that our solution is space optimal.

```
function sup(T) is of each of them is now a set of values
(S1')   let $\langle r, \ell evel, cf\ell, valset\rangle$ be max(T);
(S2')   let $vals(T)$ be $\{v \mid \langle r, -, -, valset\rangle \in T \ \wedge\ v \in valset\}$;
(S3)    let $conf\ell ict1(T)$ be $\exists\ \langle r, -, conflict, -\rangle \in T$;
(S4')   let $conf\ell ict2(T)$ be $|vals(T)| > x$;
(S5)    let $conf\ell ict(T)$ be $conf\ell ict1(T) \vee conf\ell ict2(T)$;
(N)     if $conf\ell ict(T)$ then $vals'(T) \leftarrow valset$
                  else $vals'(T) \leftarrow$ the set of the (at most) $x$ greatest values in $vals(T)$ end if;
(S6')   return$\big(\langle r, \ell evel, conf\ell ict(T), vals'(T)\rangle\big)$.
```

Figure 4 Function sup() suited to x-obstruction-freedom.

7 From Obstruction-Freedom to x-Obstruction-Freedom

This section extends the base algorithm to obtain an algorithm that solves the x-obstruction-free (n, k)-set agreement problem.

Notion of x-obstruction-freedom. This progress condition, introduced in [30], is a natural generalization of obstruction-freedom, which corresponds to the case where $x = 1$. It guarantees that for every set of processes P, with $|P| \leq x$, every correct process in P returns from its operation invocation if no process outside P takes steps for a "long enough" period of time. It is easy to see that x-obstruction-freedom and wait-freedom are equivalent in any n-process system where $x \geq n$. Differently, when $x < n$, x-obstruction-freedom depends on the concurrency pattern while wait-freedom does not.

On the value of x. We assume that $x \leq k$. Such an assumption follows from the impossibility result stating that (n, k)-set agreement cannot be wait-free solved for $n > k$, when any number of processes may crash [4, 18, 27].

Termination. In regard to x-obstruction-freedom, the Validity, Agreement and SV-Termination properties defining obstruction-free (n, k)-set agreement are the same as the ones stated in Section 2.2. The OB-Termination property now becomes:

- x-OB-termination. If there is a time after which at most x correct processes execute concurrently, each of these processes eventually decides a value.

The shared memory. To cope with the x-concurrency allowed by obstruction-freedom, that is the fact that up to x processes may compete, the array REG is such that it has now $m = n - k + x$ entries. At core, this modification in the size of the array comes from the fact that the algorithm terminates in more scenarios than the the ones covered with obstruction-freedom.

Content of a quadruplet. In the base algorithm, the four fields of a quadruplet X are a round number $X.rd$, a level $X.\ell v\ell$, a conflict value $X.cf\ell$, and a value $X.val$. Coping with x-concurrency requires to replace the last field, which was initially a singleton, with a set of values denoted hereafter $X.valset$.

Modifying function sup(). Coping with x-concurrency requires also to adapt function sup(). We describe its novel definition at Figure 4. The lines that are modified (with respect to the base definition) are followed by a "prime". We also add a new line (marked N). In detail, our modifications are the following.

```
operation propose($v_i$) is
(01) $Q \leftarrow \langle 1, \mathtt{down}, \mathtt{false}, \{v_i\}\rangle$;
(02) repeat forever
(03)     $view \leftarrow REG$.snapshot();
(04)     case $(\exists r > 0, valset : \forall x : view[x] = Q = \langle r, \mathtt{up}, \mathtt{false}, valset\rangle)$ then
                  return any value in $valset$
(05)          $(\exists r > 0, valset : \forall x : view[x] = Q = \langle r, \mathtt{down}, \mathtt{false}, valset\rangle)$ then
                  $Q \leftarrow \langle r+1, \mathtt{up}, \mathtt{false}, valset\rangle$; $REG$.write$(1, Q)$
(06)          $(\exists r > 0, valset, \ell evel : \forall x : view[x] = Q = \langle r, \ell evel, \mathtt{true}, valset\rangle)$ then
                  let $v$ be any value in $valset$;
                  $Q \leftarrow \langle r+1, \mathtt{down}, \mathtt{false}, \{v\}\rangle$;
                  $REG$.write$(1, Q)$;
(07)          otherwise let $Q \leftarrow$ sup$(view[1], \cdots, view[n], Q)$;
(08)                  $x \leftarrow$ smallest index such that $view[x] \neq Q$;
(09)                  $REG$.write$(x, Q)$
(10)     end case
(11) end repeat.
```

Figure 5 Anonymous x-obstruction-free Consensus.

- Line S1'. The last field of a quadruplet is now a set of values, denoted *valset*. Regarding the lexicographical ordering, we order the sets *valset* as follows. We order first by size, then sets of the same size are ordered from their greatest to their smallest element.
- Line S2'. The set $vals(T)$ is now the union of all the *valset* associated with the greatest round number appearing in T.
- Lines S3 and S5: We do not modify these lines.
- Line S4'. $conf\ell ict2(T)$ is modified to take into account x-concurrency. A conflict is now discovered when more than x (instead of 1) values are associated with the round number of the maximal element of T.
- New line N. The set $vals'(T)$ is equal to *valset* if $conflict(T) = \mathtt{true}$. Otherwise, it contains the (at most) x greatest values of $vals(T)$.
- Line S6'. The quadruplet returned by sup(T) differs from the one computed in Figure 2, and its last field is now the set $vals'(T)$.

Solving x-Obstruction-free (n,k)-set agreement. Figure 5, describe our x-obstruction-free (n,k)-set agreement solution. We now present its internals, detailing the key differences with our base construction described at Figure 2.

- The relation "$\sqsupset$" introduced in Section 4.1 is extended to take into account the fact that the last field of a quadruplet is now a non-empty set of values. It becomes:

$$X \sqsupset Y \overset{def}{=} (X > Y) \wedge [(X.rd > Y.rd) \vee (X.cf\ell) \vee (\neg Y.cf\ell \wedge X.valset \supseteq Y.valset)].$$

- Each process p_i maintains a local quadruplet denoted Q, and containing the last quadruplet it has computed. Initially, Q is equal to $\langle 1, \mathtt{down}, \mathtt{false}, \{v_i\}\rangle$ (line 01). This quadruplet allows its owner p_i to have an order on the quadruplets it champions during the execution of propose(v_i): if p_i champions Q at time τ, and champions Q' at time $\tau' \geq \tau$, we have $Q' \sqsupseteq Q$. This is to ensure x-OB-termination.

- The meaning of the three predicates at lines 04-06, is the following. All entries of $view$ are the same and are equal to Q, where the content of Q is either $\langle r, \texttt{up}, \texttt{false}, valset\rangle$, or $\langle r, \texttt{down}, \texttt{false}, valset\rangle$, or $\langle r, \ell evel, \texttt{true}, valset\rangle$. Hence, according to the terminology of the proof of our base construction (see Section 4.1), $view$ is homogeneous, i.e., $view$ is $\mathcal{H}(Q)$, where Q obeys to some predefined pattern.
- Lemma 15 needs to be re-formulated to take into account the last field of a quadruplet. It becomes:

  $$[(REG^{\tau} \text{ is } \mathcal{H}(X)) \wedge (X.\ell v\ell = \texttt{up}) \wedge (\neg X.cf\ell) \wedge (REG^{\tau'} \text{ is } \mathcal{H}(Y)) \wedge (Y.rd \geq X.rd)] \Rightarrow (Y.valset \supseteq X.valset \vee X.valset \supseteq Y.valset).$$

  The lemma is true as soon as the number of participating processes does not exceed the number of available registers in REG.
- For the agreement property, we have to show that $(n-k+x)$ registers are sufficient. Our reasoning is similar to the one depicted in Section 5. More precisely, let us consider the $(k-x)$ first decided values, where the notion "first" is defined with respect to the linearization time of the snapshot invocation (line 03) that immediately precedes the invocation of the corresponding deciding statement (`return()` at line 04). Let τ be the time just after the linearization of these $(k-x)$ "deciding" snapshots. Starting from τ, at most $(n-(k-x)) = (n-k+x)$ processes access the array REG, which is made up of exactly $(n-k+x)$ registers. Consider the $(k-x+1)$-th deciding snapshot, let it be at $\tau' > \tau$. According to the precondition at line 04, $REG^{\tau'}$ is $\mathcal{H}(X)$ for some X with $X.\ell v\ell = \texttt{up}$ and $X.cf\ell = \texttt{false}$. Observe that in such case $|X.valset| \leq x$.
  According to the new statement of Lemma 15, since starting from τ the number of participating processes is always less than the number of registers, then all the deciding snapshots computed after τ' are associated with a set of values that is either a subset or a superset of $X.valset$. Hence, at most x values can be decided starting from τ'.
- Regarding now x-OB-termination, let us first notice that the underlying snapshot algorithm is non-blocking [15]. Hence, it ensures that, whatever the concurrency pattern is, at least one snapshot invocation always terminates. Then, the key is at line 06. When a process p_i detects a conflict ($Q.cf\ell = \texttt{true}$), it starts a new round with a singleton set. Hence, if there is a finite time after which no more than x processes are taking steps, there is a round after which at most x values survive and appear in the next rounds. From that round, no new conflict can appear, and eventually each of the (at most) x running processes obtains a snapshot entailing a decision.

8 Conclusion

In this paper, we first present a one-shot obstruction-free (n,k)-set agreement algorithm for a system made up of n asynchronous anonymous processes that communicate with atomic read/write registers. This algorithm uses only $(n-k+1)$ registers. In terms of space complexity, it is the best algorithm known so far, and in the case of consensus it is asymptotically optimal [14]. This algorithm answers the challenge posed in [8], and establishes a novel upper bound of $(n-k+1)$ on the number of registers to solve one-shot obstruction-free (n,k)-set agreement. This upper bound improves the ones stated in [10] for anonymous and non-anonymous systems.

Further, we introduce a simple extension of our base construction that solves repeated (n,k)-set agreement . The lower bound of $(n-k+1)$ atomic registers was established in [10] for this problem. Our algorithm proves that this bound is tight. Then, we detail a one-shot algorithm to solve the (n, k)-set agreement problem in the context of x-obstruction-freedom. This algorithm makes use of $(n - k + x)$ atomic read/write registers.

All our algorithms rely on the same round-based data structure. The base one-shot algorithm does not require persistent local variables, and in addition to a proposed value, an atomic register solely contains two bits and a round number. The algorithm solving repeated (n, k)-set agreement requires that each atomic register includes two additional fields.

Let us call "MWMR-*nb*" of a problem P, the minimal number of MWMR atomic registers needed to solve P in an asynchronous system of n processes. This paper shows that $(n-k+1)$ is the MWMR-*nb* of repeated obstruction-free (n, k)-set agreement. We conjecture that $(n - k + 1)$ is also the MWMR-*nb* of one-shot obstruction-free (n, k)-set agreement, and more generally that $(n - k + x)$ is the MWMR-*nb* of one-shot x-obstruction-free (n, k)-set agreement, when $1 \leq x \leq k < n$.

References

1 Afek Y., Attiya H., Dolev D., Gafni E., Merritt M., and Shavit N., Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890 (1993)

2 Aguilera M., A pleasant stroll through the land of infinitely many creatures. *ACM SIGACT news, DC column*, 35(2):36–59 (2004)

3 Attiya H., Guerraoui R., Hendler D., and Kuznetsov P., The complexity of obstruction-free implementations. *Journal of the ACM*, 56(4), Article 24, 33 pages (2009)

4 Borowsky E. and Gafni E., Generalized FLP impossibility result for t-resilient asynchronous computations. *Proc. 25-th Annual ACM Symposium on Theory of Computing (STOC'93)*, ACM Press, pp. 91–100 (1993)

5 Bouzid Z. and Raynal M. and Sutra P. Anonymous Obstruction-free (n, k)-Set Agreement with $n - k + 1$ Atomic Read/Write Registers *RR 2027, Univerité de Rennes 1*, (2015)

6 Chaudhuri S., More *Choices* Allow More *Faults:* Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation,* 105:132–158 (1993)

7 Delporte C., Fauconnier H., Gafni E., and Lamport L., Adaptive register allocation with a linear number of registers. *Proc. 27th Int'l Symposium on Distributed Computing (DISC'13)*, Springer LNCS 8205, pp. 269–283 (2013)

8 Delporte C., Fauconnier H., Gafni E., and Rajsbaum S., Black art: obstruction-free k-set agreement with |MWMR registers| < |proccesses|. *Proc. First Int'l Conference on Networked Systems (NETYS'13)*, Springer LNCS 7853, pp. 28–41 (2013)

9 Delporte C., Fauconnier H., Gafni E., and Rajsbaum S., Linear space bootstrap communication schemes. *Theoretical Computer Science*, 561:122–133 (2015)

10 Delporte C., Fauconnier H., Kuznetsov P. and Ruppert E., On the space complexity of set agreement. *Proc. 34th Int'l Symposium on Principles of Distributed Computing (PODC'15)*, ACM Press (2015)

11 Ellen Fich F., Luchangco V., Moir M., and Shavit N., Obstruction-free algorithms can be practically wait-free. *Proc. 19th Int'l Symposium on Distributed Computing (DISC'05)*, Springer LNCS 3724, pp. 78–92 (2005)

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

13 Flocchini P., Prencipe G., Santoro N., and Widmayer P., Hard tasks for weak robots: the role of common knowledge in pattern formation by autonomous mobile robots. *Proc. 10th Int'l Symposium on Algorithms and Computation (ISAAC'99)*, Springer LNCS 1741, pp. 93–102 (1999)

14 Gelashvili R., Optimal Space Complexity of Consensus for Anonymous Processes *Proc. 29th Int'l Symposium on Distributed Computing (DISC'15)*, Springer LNCS, *in press*, (2015)

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

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

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

18 Herlihy M.P. and Shavit N., The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923 (1999)

19 Herlihy M.P. and Shavit N., *The art of multiprocessor programming.* Morgan Kaufmann, 508 pages (2008) (ISBN 978-0-12-370591-4).

20 Herlihy M.P. and Wing J.M, Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492 (1990)

21 Lamport L., Concurrent reading while writing. *Communications of the ACM*, 20(11):806–811 (1977)

22 Lamport L., On interprocess communication, Part I: basic formalism. *Distributed Computing*, 1(2):77–85 (1986)

23 Loui M.C., and Abu-Amara H.H., Memory Requirements for Agreement Among Unreliable Asynchronous Processes. *Par. and Distributed Computing: vol. 4 of Advances in Comp. Research,* JAI Press, 4:163–183 (1987)

24 Merritt M. and Taubenfeld G., Computing with infinitely many processes. *Information & Computation*, 233:12–31 (2013)

25 Peterson G.L., Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5:46–55 (1983)

26 Raynal M., *Concurrent programming: algorithms, principles, and foundations.* Springer, 530 pages (2013) (ISBN 978-3-642-32026-2).

27 Saks M.S. and Zaharoglou F., Wait-Free k-Set Agreement is Impossible: The Topology of Public Knowledge. *SIAM Journal Computing* 29(5):1449–1483 (2000)

28 Suzuki I. and Yamashita M., Distributed anonymous mobile robots. *Proc. 3rd Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO'96)*, Carleton University Press, pp. 313–330 (1996)

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

30 Taubenfeld G., Contention-sensitive data structure and algorithms. *Proc. 23th Int'l Symposium on Distributed Computing (DISC'09)*, Springer LNCS 5805, pp. 157–171 (2009)

31 L Zhu, A Tight Space Bound for Consensus. Private communication (2015)