Anonymous Obstruction-free (n, k)-Set Agreement with $n - k + 1$ Atomic Read/Write Registers

Zohir Bouzid, Michel Raynal, Pierre Sutra

Publications Internes de l'IRISA
ISSN : 2102-6327
PI 2027 – January 2015

Anonymous Obstruction-free (n, k)-Set Agreement with $n - k + 1$ Atomic Read/Write Registers

Zohir Bouzid[*] Michel Raynal[**] [***] Pierre Sutra[****]

Abstract: The k-set agreement problem is a generalization of the consensus problem. Namely, assuming each process proposes a value, each non-faulty process has to decide a value such that each decided value was proposed, and no more than k different values are decided. This is a hard problem in the sense that it cannot be solved in asynchronous systems as soon as k or more processes may crash. One way to circumvent this impossibility consists in weakening its termination property, requiring that a process terminates (decides) only if it executes alone during a long enough period. This is the well-known obstruction-freedom progress condition.

Considering a system of n *anonymous asynchronous* processes, which communicate through atomic *read/write registers only*, and where *any number of processes may crash*, this paper addresses and solves the challenging open problem of designing an obstruction-free k-set agreement algorithm with $(n - k + 1)$ atomic registers only. From a shared memory cost point of view, this algorithm is the best algorithm known so far, thereby establishing a new upper bound on the number of registers needed to solve the problem (its gain is $(n-k)$ with respect to the previous upper bound). The algorithm is then extended to address the repeated version of (n, k)-set agreement. As it is optimal in the number of atomic read/write registers, this algorithm closes the gap on previously established lower/upper bounds for both the anonymous and non-anonymous versions of the repeated (n, k)-set agreement problem. Finally, for $1 \leq x \leq k < n$, a generalization suited to x-obstruction-freedom is also described, which requires $(n - k + x)$ atomic registers only.

Key-words: Anonymous processes, Asynchronous system, Atomic read/write register, Bounded number of registers, Consensus, Distributed algorithm, Distributed computability, Fault-tolerance, k-Set agreement, Obstruction-freedom, Process crash, Repeated k-set agreement, Upper bound.

Accord k-ensembliste asynchrone et anonyme avec $(n - k + 1)$ registres atomiques

Résumé : *Cet article présente un algorithme asynchrone qui résoud l'accord k-ensenbliste dans un système de n processus asynchrones et anonymes communiquant via $(n-k+1)$ registres atomiques du type lire/écrire, et dans lequel un nombre quelconque d'entre eux peut s'arrêr de fçon inopinée (crash failure). La propriété de vivacité garantie par l'algorithme est appelée "*obstruction-freedom*".*

Mots clés : *Accord k-ensembliste, Borne de complexité, Consensus, système asynchrone, système anonyme, registres atomiques read/write, crash de processus, calcul distribué, tolérance aux fautes.*

---

[*] ASAP : équipe commune à l'IRISA (Université de Rennes 1) et Inria
[**] Institut Universitaire de France
[***] ASAP : équipe commune à l'IRISA (Université de Rennes 1) et Inria
[****] University of Neuchâtel, Switzerland

1 Introduction

A first challenge: cope with multi-writer atomic registers Pioneering works (such as [21, 25]) have shown that processes have to cope not only with finite asynchrony (finite but arbitrary process speed) but also with infinite asynchrony (process crash failures), a context in which mutex-based synchronization mechanisms become useless. This approach has promoted the design of concurrent algorithms as a central topic of *fault-tolerant* distributed computing. See for example Herlihy's seminal paper [16], or recent textbooks such as [19, 26, 29].

When processes may communicate with *Single-Writer Multi-Reader* (SWMR) atomic registers, a concurrent algorithm usually associates an SWMR register with each process. This type of registers allows any process to give information to all the other processes by writing in its own register, and obtain information from them by reading their SWMR registers. The classical snapshot algorithm introduced in [1] is a well-known example of use of such atomic registers.

When processes communicate with *Multi-Writer Multi-Reader* (MWMR) atomic registers, the situation is different. As any process can write any register, the previous association is no longer given for free. An approach to cope with such registers consists in emulating SWMR registers on top of MWMR registers, and then benefit from existing SWMR-based algorithms. It is shown in [6, 8] that, in a system of n processes, (a) $(2n-1)$ MWMR atomic registers are needed to "wait-free" simulate one SWMR atomic register, and (b) only n MWMR atomic registers are needed if the simulation is required to be only "non-blocking"[1].

This simulation approach becomes irrelevant if the underlying system provides the n processes with less than n atomic MWMR registers. So, we focus here on what we name *genuine* concurrent algorithms, where "genuine" means "without simulating SWMR registers on top of MWMR registers". An important question is then "Given a problem, how many MWMR atomic registers are needed to solve it with a genuine algorithm?" Unfortunately, as stressed in [7], the design of genuine algorithms based on MWMR atomic registers is still in its infancy, and sometimes resembles "black art" in the sense that their underlying intuition is difficult to capture and formulate.

A second challenge: cope with anonymous processes In some algorithms based on MWMR atomic registers, a process is required to write a pair made up of the data value it wants to write, plus control values, those including its identity. This is for example the case of snapshot algorithms based on MWMR atomic registers [26].

So, a second question that comes to mind is: "Is it possible to solve a given problem with MWMR atomic registers and *anonymous* processes; moreover, if the answer is "yes", how many registers are needed?" To be more precise, let us recall that, in an anonymous system, processes have no identity, have the same code, and the same initialization of their local variables. It is common to remind that, due to privacy motivations, anonymous systems are becoming more and more important.

Consensus and k-set agreement The paper considers the k-set agreement problem in a system of n processes. This problem, introduced in [5], and denoted (n,k)-set agreement in the following, is a generalization of consensus, which corresponds to the instance where $k=1$. Assuming each participating process proposes a value, each non-faulty process must decide a value (termination), which was proposed by some process (validity), and at most k different values can be decided (agreement).

Impossibility results and the obstruction-freedom progress condition It is well-known that it is impossible to design a deterministic wait-free consensus algorithm in asynchronous systems prone to even a single crash failure, be the underlying communication medium an asynchronous send/receive network [12], or a set of read/write atomic registers [23]. It is also shown in [4, 18, 27] that, if k or more processes may crash, there is no deterministic wait-free read/write algorithm that can solve (n,k)-set agreement.

As we are interested in the computing power of *pure read/write* asynchronous systems, we want to neither enrich the underlying system with additional power such as synchrony assumptions, random numbers, or failure detectors, nor impose constraints restricting the input vector collectively proposed by the processes. So, we consider here a progress condition weaker than wait-freedom, named *obstruction-freedom* [17]. In the consensus or (n,k)-set agreement context, obstruction-freedom requires a process to decide a value only if it executes solo during a "long enough period" (which means that, during this period, it is not bothered by other processes). An in-depth study of complexity issues of obstruction-free algorithms is presented in [3].

Several obstruction-free consensus algorithms suited to non-anonymous systems have been proposed (e.g., [7, 11] to cite a few). When considering anonymous systems, the obstruction-free algorithm presented in [15] requires $(8n+2)$ MWMR atomic registers to solve consensus, and the obstruction-free algorithms described in [7, 9] solve (n,k)-set agreement with $2(n-k)+1$ underlying MWMR atomic registers.

[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].

Motivation and content of the paper This paper presents 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. This algorithm (called *base algorithm* in the following) requires $(n - k + 1)$ MWMR atomic registers (i.e., exactly n registers when one is interested in the consensus problem).

It is shown in [10] that $\Omega(\sqrt{n})$ MWMR atomic registers is a lower bound for obstruction-free consensus. This lower bound has recently been generalized to $\Omega(\sqrt{\frac{n}{k} - 2})$ for (n, k)-set agreement in anonymous systems [9]. On another hand, and as already pointed out, the best obstruction-free (n, k)-set agreement algorithm known so far requires $2(n - k) + 1$ MWMR registers [7, 9]. Hence, the base algorithm proposed in this paper provides us with a gain of $2(n - k) + 1 - (n - k + 1) = (n - k)$ MWMR atomic registers.

In the *repeated* version of the (n, k)-set agreement problem, the processes participate in a sequence of (n, k)-set agreement instances. It is shown in [9] that $(n - k + 1)$ atomic registers are necessary to solve repeated (n, k)-set agreement, be the system anonymous or non-anonymous. The present paper shows that a simple modification of the base obstruction-free (n, k)-set agreement algorithm solves the *repeated* (n, k)-set agreement problem without requiring additional atomic registers. It follows that, as this algorithm requires $(n - k + 1)$ atomic registers, it is optimal, which closes the gap on previous proposed upper bounds for the repeated (n, k)-set agreement problem.

To attain its goal, the proposed base algorithm, which is round-based, follows the execution pattern "snapshot; local computation; write", where the snapshot and the write are on the $(n - k + 1)$ MWMR atomic registers. This pattern is reminiscent of the one called "look; compute; move" introduced in [13, 28] in the context of robot algorithms. Interestingly, no process needs to maintain local information between successive rounds. In this sense, the algorithm is *locally memoryless*.

From a more technical point of view, each atomic register contains a quadruplet consisting of a round number, two control bits, and a proposed value (whose size depends only on the application). The algorithm exploits a partial order on the quadruplets that are written into MWMR atomic registers. The way each process computes new quadruplets is the key of the algorithm. (The extended version for repeated (n, k)-set agreement, requires sixuplets.)

Roadmap The paper is composed of 8 sections. Section 2 presents the computing model and definitions used in the paper. The presentation is done incrementally. First, Section 3 presents the base obstruction-free algorithm solving consensus. This algorithm captures the essence of the solution. It is proved correct in Section 4. Then, Section 5 extends this base algorithm to obtain an anonymous obstruction-free algorithm solving (n, k)-set agreement, and Section 6 addresses the case where (n, k)-set agreement is used repeatedly. Section 7 extends the base algorithm to the x-obstruction-freedom progress condition (only $(n - k + x)$ registers are then required by the algorithm). Finally, Section 8 concludes the paper.

2 Computation Model and Obstruction-free Consensus

2.1 Computing Model

Process model The system is composed of n asynchronous processes, denoted p_1, ..., p_n. When considering a process p_i, the integer i is called its index. Indexes are used to facilitate the exposition from an external observer point of view. Processes do not have identities and have the very same code. We assume that they know the value n.

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].

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 write or a read of an atomic register (access to the shared memory) possibly followed by a finite number of internal operations (on the local variables of the process that issued the operation).

Communication model In addition to processes, the computing model includes a communication medium made up of m atomic multi-writer/multi-reader (MWMR) atomic registers[3]; the value of m depends on the problem we want to solve. These registers are encapsulated in an array denoted $REG[1..m]$.

"*Atomic*" means that the read and write operations on a register $REG[x]$, $1 \leq x \leq m$, 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 any concurrent object defined from a sequential

[2]No process knows if it is correct or faulty. This is because, before crashing, a faulty process behaves as a correct process.

[3]Let us notice that the anonymity assumption prevents processes from using single-writer/multi-reader registers.

specification, atomicity is called *linearizability* [20]. More generally, the sequence of operations is called a *linearization*, and the time instant at which an operation appears as being executed is called its *linearization point*.

From atomic registers to a snapshot object At the upper layer (where consensus or (n, k)-set agreement is solved), the array $REG[1..m]$ is used to define a snapshot object [1]. This object, denoted REG, provides the processes with two operations denoted 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 value of the whole array. The snapshot object is atomic (see above), which means that each invocation of REG.snapshot() appears as if it executed instantaneously. Hence, at this observation level, 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] (for completeness this algorithm is presented in Appendix A). This implementation does not require additional atomic registers. In the following we consider that this snapshot abstraction is supplied by this underlying layer.

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 these 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 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 where the agreement property is:

- 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 stated in [5].

3 Obstruction-free Anonymous Consensus Algorithm

The algorithm is described 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 $\langle 0, \texttt{down}, \texttt{false}, \perp\rangle$. 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 $\{\texttt{up}, \texttt{down}\}$, where $\texttt{up} > \texttt{down}$.
- The third field, denoted $cf\ell$ (conflict), is a Boolean (init to false). We assume $\texttt{true} > \texttt{false}$.
- 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 the default value $\perp$ is smaller than any of them.

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

[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   % T is a set of quadruplets %
(S1) let ⟨r, levelℓ, −, v⟩ be max(T);                % lexicographical order %
(S2) let vals(T) be {w | ∃⟨r, −, −, w⟩ ∈ T};
(S3) let conflict1(T) be ∃ ⟨r, −, true, −⟩ ∈ T;      % conflict inherited %
(S4) let conflict2(T) be |vals(T)| > 1;              % conflict discovered %
(S5) let conflict(T) be conflict1(T) ∨ conflict2(T);
(S6) return(⟨r, levelℓ, conflict(T), v⟩).
```

Figure 1: The function sup()

The notion of a conflict and the function sup() The function sup(), defined in Figure 1, plays a central role in the obstruction-free (n, k)-agreement 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 $\langle r, level\ell, -, v\rangle$ be the maximal element of T according to lexicographical ordering (line S1), and $vals(T)$ 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).

- There is a quadruplet $X = \langle r, -, \texttt{true}, -\rangle$ 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 $X.\ell v\ell = \texttt{true}$. 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 contain different values (i.e., $X.val \neq Y.val$) (lines S2 and S4). In this case we say say that the conflict is "discovered".

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

```
operation propose(v_i) is
(01) repeat forever
(02)   view ← REG.snapshot();
(03)   case (∀x : view[x] = ⟨r, up, false, val⟩      where r > 0) then return(val)
(04)        (∀x : view[x] = ⟨r, down, false, val⟩    where r > 0) then REG.write(1, ⟨r + 1, up, false, val⟩)
(05)        (∀x : view[x] = ⟨r, level, true, val⟩    where r > 0) then REG.write(1, ⟨r + 1, down, false, val⟩);
(06)        otherwise let ⟨r, level, cfℓ, val⟩ ← sup(view[1], · · · , view[n], ⟨1, down, false, v_i⟩);
(07)                  x ← smallest index such that view[x] ≠ ⟨r, level, cfℓ, val⟩;
(08)                  REG.write(x, ⟨r, level, cfℓ, val⟩)
(09)   end case
(10) end repeat.
```

Figure 2: Anonymous obstruction-free Consensus

The algorithm The algorithm is pretty simple. It consists in an appropriate management of the snapshot object REG, so that the n quadruplets it contains (a) never allow validity and 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).

When a process p_i invokes proposes(v_i), it enters a loop that it will exit at line 03 (if it terminates), by executing the statement return$(val\ell)$, where $val\ell$ is the value it decides.

After entering the loop a process issues first a snapshot, and assigns the returned array to its local variable $view[1..n]$ (line 01). Then, there are two main cases according to the value of $view$.

- Case 1 (lines 03-05). All entries of $view_i$ contain the same quadruplet $\langle r, level\ell, conflictval\rangle$, and $r > 0$.
  There are three sub-cases.
  - Case 1.1. If the level is up and the conflict is false, the invoking process decides the value $val\ell$ (line 03).
  - Case 1.2. If the level is down and the conflict field is false, the invoking process decides the value $val\ell$ (line 03). is false, process p_i enters the next round by writing $\langle r + 1, \texttt{up}, \texttt{false}, val\rangle$ in the first entry of REG (line 04).
  - Case 1.3. If there is a conflict, p_i enters the next round by writing $\langle r + 1, \texttt{down}, \texttt{false}, val\rangle$ in the first entry of REG (line 05).
- Case 2 (lines 06-08). Not all entries of $view_i$ are equal or one of them contains $\langle 0, -, -, -\rangle$.
  In this case, process p_i calls the internal function sup$(view[1], \cdots, view[n], \langle 1, \texttt{down}, \texttt{false}, v_i\rangle)$ (line 06), which returns a quadruplet X that is greater than all the input quadruplets or equal to the greatest of them. As we have seen, 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 the function $\texttt{sup}()$, $X.val$ cannot be $\perp$.

Let us notice that, as none of the predicates of lines 03-05 is satisfied, not all entries of $view[1..n]$ can be equal to the previous quadruplet X. The invoking process p_i writes X into $REG[x]$, where, from its point of view, x is the first entry of REG whose content is different from X (lines 07-08).

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

Let us now consider the case where, while p_i is executing, another process p_j invokes $\texttt{propose}(v_j)$ with $v_j = v_i$. It is easy to see that p_i and p_j collaborate then to fill in REG with the same quadruplet $\langle 2, \texttt{up}, v_i\rangle$. 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 number.

Remark 1 Let us notice that no process needs to memorize in its local memory values that will be used in the next round. Not only the processes are anonymous, but their code is 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 If $N > n$, the number of proposed values can be greater than the upper bound n on the concurrency degree. This versatility dimension of the algorithm is a direct consequence of the previous "locally memoryless" property.

4 Proof of the Algorithm

After a few definitions provided in Section 4.1, Section 4.2 shows that the relation "$\sqsupseteq$" defined on quadruplets is a partial order. This relation is central to prove properties of the algorithm. Such properties are stated and proved in Sections 4.3 and 4.4. Based on these previous properties, Section 4.5 establishes the correctness of our algorithm.

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 $X.val$, respectively, and $>$ and $\geq$ refer to the classical lexicographical order on $\mathcal{E}$. Moreover, where appropriate, an array $view[1..n]$ is considered as the set $\{view[1], \cdots, view[n]\}$.

Definition 1 *let* $X, Y \in \mathcal{E}$.

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

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 of this total order, and is consequently the key cornerstone on which relies the proof of our algorithm.

When $X \sqsupset Y$, we say "X *strictly dominates* Y". X *dominates* Y, denoted $X \sqsupseteq Y$, if $(X \sqsupset Y)$ or $(X = Y)$ holds. The relations $\sqsubset$ and $\sqsubseteq$ are defined in the natural way.

Definition 2 *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 xxx of a process p_i is denoted xxx_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 *Let $\mathcal{W}(x,X)$ denote the writing of a quadruplet X in the register $REG[x]$.*

Definition 3 *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, at time τ, p_j covers $REG[x]$, 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 1 $((X \sqsupset Y \sqsupset Z) \wedge (X.rd = Y.rd = Z.rd)) \Rightarrow (X.cf\ell \vee (\neg Z.cf\ell \wedge X.va\ell = Z.va\ell))$.

Proof Let us assume that $(\neg\ X.cf\ell)$ holds, we have to prove $\neg Z.cf\ell$ and $X.va\ell = Z.va\ell$. It then follows from the lemma assumption and the definition of $\sqsupset$ that we have:

$$((X \sqsupset Y) \wedge (X.rd = Y.rd) \wedge (\neg X.cf\ell)) \Rightarrow (\neg Y.cf\ell \wedge X.va\ell = Y.va\ell).$$

Hence we can use the same argument as above to show that $(\neg Z.cf\ell \wedge Y.va\ell = Z.va\ell)$:

$$((Y \sqsupset Z) \wedge (Y.rd = Z.rd) \wedge (\neg Y.cf\ell)) \Rightarrow (\neg Z.cf\ell \wedge Y.va\ell = Z.va\ell).$$

Summarizing we have $(\neg Z.cf\ell \wedge X.va\ell = Z.va\ell)$. This proves the claim. $\square_{Lemma\ 1}$

Lemma 2 *$\sqsupseteq$ is a partial order.*

Proof To prove the transitivity property, let us assume that $X \sqsupseteq Y$ and $Y \sqsupseteq Z$. We have to show that $X \sqsupseteq Z$. If $X = Y$ or $Y = Z$, the claim follows trivially. Hence, let us assume that Y is neither X nor Z. As $(X \sqsupset Y) \Rightarrow (X > Y)$, $(Y \sqsupset Z) \Rightarrow (Y > Z)$, it follows that $X > Z$. To prove $X \sqsupset Z$, it remains to show that $((X.rd > Z.rd) \vee (X.cf\ell) \vee (\neg Z.cf\ell \wedge X.va\ell = Z.va\ell))$. Let us observe that, due to the definition of $\sqsupset$, we have $(X \sqsupset Y) \Rightarrow \big((X.rd > Y.rd) \vee (X.cf\ell) \vee (\neg Z.cf\ell \wedge X.va\ell = Y.va\ell)\big)$. There are three cases.

- Case $(X.rd > Y.rd)$. As $Y \sqsupset Z$ we have $(Y.rd \geq Z.rd)$. Hence, $(X.rd > Z.rd)$.
- Case $(X.rd = Y.rd) \wedge (Y.rd > Z.rd)$. Then, we have $(X.rd > Z.rd)$.
- Case $(X.rd = Y.rd) \wedge (Y.rd = Z.rd)$. Then, Lemma 1 $\Rightarrow (X.cf\ell \vee (\neg Z.cf\ell \wedge X.va\ell = Z.va\ell))$.

In each case, the transitivity property follows.

To prove the antisymmetry property, we show that if $X \sqsupset Y$ then $Y \not\sqsupset X$. Assume for contradiction that $X \sqsupset Y$ and $Y \sqsupset X$. It follows that $X > Y$ and $Y > X$, contradiction. $\square_{Lemma\ 2}$

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

The definition of $\mathsf{sup}()$ appears in Figure 1.

Lemma 3 *Let T be a set of quadruplets. For every $X \in T$: $\mathsf{sup}(T) \sqsupseteq X$.*

Proof Let $X \in T$ and $S = \mathsf{sup}(T)$. We have to prove that $S \sqsupseteq X$. Let us first observe that, as $S = \mathsf{sup}(T) \geq \mathsf{max}(T) \geq X$, we have $S \geq X$. If $S = X$ then the lemma follows immediately. So let us assume in the following that $S > X$. There are two cases.

- If $S.rd > X.rd$, then $S \sqsupset X$, and the lemma follows.

- Assume that $S.rd = X.rd$. We need to show that $(S.cf\ell) \vee (\neg X.cf\ell \wedge S.va\ell = X.va\ell)$.

  In the following we prove that $(\neg S.cf\ell \Rightarrow \neg X.cf\ell)$. Therefore we need then only to show that $(S.cf\ell) \vee (S.va\ell = X.va\ell)$.

  Let us first prove $(\neg S.cf\ell \Rightarrow \neg X.cf\ell)$. We do it by proving the contrapositive $X.cf\ell \Rightarrow S.cf\ell$. If $(X.cf\ell)$, we have the following. Since $X.rd = S.rd = \mathsf{sup}(T).rd$, it follows that the predicate $conf\ell ict1(T)$ is true, which implies that $S.cf\ell = \mathsf{sup}(T).cf\ell$ is also true. Therefore $X.cf\ell \Rightarrow S.cf\ell$.

  Let us now show the second part, i.e., either $(S.cf\ell)$ or $(S.va\ell = X.va\ell)$ holds. Assume that $(S.va\ell \neq X.va\ell)$ and let us prove that $(S.cf\ell)$ is true. Let us observe that, due to the definition of $S = \mathsf{sup}(T)$ (Figure 1), $\mathsf{max}(T).va\ell = \mathsf{sup}(T).va\ell = S.va\ell$. But we assumed $S.va\ell \neq X.va\ell$. Therefore $\mathsf{max}(T).va\ell \neq X.va\ell$. This means that there are at least two elements in T, namely X and $\mathsf{max}(T)$, which are associated with the maximal round $S.rd$, and which carry distinct values ($X.va\ell \neq \mathsf{max}(T).va\ell$). Hence, the predicate $conf\ell ict2(T)$ is satisfied, and consequently $\mathsf{sup}(T).cf\ell$ is equal to `true`. Therefore $S = \mathsf{sup}(T) \sqsupset X$.

  $\square_{Lemma\ 3}$

Lemma 4 *If p_i executes $\mathcal{W}(-, Y)$ at time τ, then for every $X \in view_i^\tau : Y \sqsupseteq X$.*

Proof We consider two cases according to the line at which the write occurs.

- Y is written at line 04 or 05. It follows that $Y.rd = (\mathsf{max}(view_i^\tau).rd) + 1$. Therefore, for every $X \in view_i^\tau : Y.rd > X.rd$. Hence $Y \sqsupset X$.
- Y is written at line 08. In this case, due to the invocation of the function $\mathsf{sup}()$ at line 06, the value Y written by p_i is equal to $\mathsf{sup}(T)$ where $T = \{view_i^\tau[1], \cdots, view_i^\tau[n], \langle 1, \mathtt{down}, \mathtt{false}, v_i\rangle\}$. According to Lemma 3, it follows that for every $X \in view_i^\tau$ we have $Y = \mathsf{sup}(T) \sqsupseteq X$.

$\square_{Lemma\ 4}$

Lemma 5 *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 4, $X \sqsupseteq view_i^{\tau_2}[x]$. 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 a 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 2), we obtain $X \sqsupseteq REG^\tau[x]$, a contradiction that concludes the proof of the lemma. $\square_{Lemma\ 5}$

Lemma 6 *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 $Y \sqsupseteq X$.*

Proof If $REG^{\tau'} = REG^\tau$, the lemma holds trivially. So let us assume in the following that $REG^{\tau'} \neq REG^\tau$ which means that a write happens between τ and τ'. If $\langle 0, \mathtt{down}, \mathtt{false}, \perp\rangle \in REG^\tau$, as every quadruplet Y written in REG is such that $Y.rd \geq 1$ (line 04, 05, or lines 06-08), we have $Y \sqsupset \langle 0, \mathtt{down}, \mathtt{false}, \perp\rangle$.

So, let us assume that $\langle 0, \mathtt{down}, \mathtt{false}, \perp\rangle \notin REG^\tau$ and consider the last write in REG before τ. Assume this happens at $\tau^- \leq \tau$ and let p_i be the writing process. Process p_i has no write covering a register of REG at time τ^-. Consequently, at most $(n-1)$ processes[5] have a write covering a register of REG at time τ^-. Hence, there exists $x \in \{1, \ldots, n\}$ such that no write is covering $REG[x]$ at time τ^-. Let $X = REG^{\tau^-}[x] = REG^\tau[x]$. If $X = Y$ then the claim of the lemma follows trivially. So

[5]Let us notice that this is the only place in the proof where the consensus version of the algorithm requires more than $(n-1)$ MWMR atomic registers.

assume in the following that $X \neq Y$. Since $REG^{\tau^-}[x] = X$, $REG^{\tau'}[x] = Y$ and $Y \neq X$, there is necessarily a write $\mathcal{W}(x,Y)$ that occurred between τ^- and τ'. As this write was not covering a register of REG at time τ^-, it follows (according to Lemma 5) that $Y \sqsupseteq X$, which proves the lemma. $\square_{Lemma\ 6}$

The following two lemmata are corollaries of Lemma 6.

Lemma 7 *If REG^{τ} is $\mathcal{H}(X)$, $REG^{\tau'}$ is $\mathcal{H}(Y)$, and $\tau' \geq \tau$, then $Y \sqsupseteq X$.*

Lemma 8 *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 $(Y.va\ell = X.va\ell)$.*

Proof According to Lemma 7, $Y \sqsupseteq X$. If $Y = X$ then the claim follows immediately. So let us assume $Y \sqsupset X$. As $(Y.rd = X.rd)$ and $(\neg Y.cf\ell)$, the definition of $\sqsupseteq$ implies that $Y.va\ell = X.va\ell$. $\square_{Lemma\ 8}$

4.4 Exploiting homogeneous snapshots

Lemma 9 $[(X \in REG^{\tau}) \wedge (X.\ell v\ell = \mathtt{up})] \Rightarrow \big(\exists\, \tau' < \tau$*: $REG^{\tau'}$ is $\mathcal{H}(Z)$, where* $Z = \langle X.rd - 1, \mathtt{down}, \mathtt{false}, X.va\ell\rangle\big)$.

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

- If $X.cf\ell = \mathtt{false}$, then let $X' = X$. Since $X.\ell v\ell = X'.\ell v\ell = \mathtt{up}$, X was necessarily written into REG by some process (let us remember that the initial value of each register of REG is $\langle 0, \mathtt{down}, \mathtt{false}, \bot\rangle$).
- If $X.cf\ell = \mathtt{true}$, let us consider the time τ_1 at which X was written for the first time into REG, say by p_i. Since $X.\ell v\ell = \mathtt{up}$, 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 $X.cf\ell = \mathtt{false}$).

  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 show first 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.

It follows then from the precondition of line 04 that $view^{\tau_2}$ is $\mathcal{H}(\langle X'.rd - 1, \mathtt{down}, \mathtt{false}, X'.va\ell\rangle)$. Hence, the lemma follows. $\square_{Lemma\ 9}$

Lemma 10 $[(REG^{\tau}$ *is* $\mathcal{H}(X)) \wedge (X.\ell v\ell = \mathtt{up}) \wedge (\neg X.cf\ell) \wedge (REG^{\tau'}$ *is* $\mathcal{H}(Y)) \wedge (Y.rd \geq X.rd)] \Rightarrow (Y.va\ell = X.va\ell)$.

Proof The proof is by induction on $Y.rd$. Let us first assume that $Y.rd = X.rd$, for which we consider two cases.

- Case 1: $\tau \geq \tau'$. Since $X.cf\ell = \mathtt{false}$, it follows according to Lemma 8 that $Y.va\ell = X.va\ell$.
- Case 2: $\tau' > \tau$. According to Lemma 7, $Y \sqsupseteq X$. As $Y.rd = X.rd$, it follows that $Y.\ell v\ell \geq X.\ell v\ell = \mathtt{up}$, and consequently $Y.\ell v\ell = \mathtt{up}$.

  Summarizing we have $REG^{\tau'}$ is $\mathcal{H}(Y)$, $Y.\ell v\ell = \mathtt{up}$ and $Y.rd = X.rd$. According to Lemma 9, This implies that it exists $\tau_1 < \tau$ and $\tau'_1 < \tau'$ such hat REG^{τ_1} is $\mathcal{H}(\langle X.rd - 1, \mathtt{down}, \mathtt{false}, X.va\ell\rangle)$ and $REG^{\tau'_1}$ is $\mathcal{H}(\langle Y.rd - 1, \mathtt{down}, \mathtt{false}, Y.va\ell\rangle)$. According to Lemma 7, we have either $\langle X.rd - 1, \mathtt{down}, \mathtt{false}, X.va\ell\rangle \sqsupseteq \langle Z.rd - 1, \mathtt{down}, \mathtt{false}, Y.va\ell\rangle$ or $\langle Y.rd - 1, \mathtt{down}, \mathtt{false}, Y.va\ell\rangle \sqsupseteq \langle X.rd - 1, \mathtt{down}, \mathtt{false}, X.va\ell\rangle$. Since by assumption $X.rd = Y.rd$, it follows that $X.va\ell = Y.va\ell$. The contradiction establishes the claim.

For the induction step, let assume that the lemma is true up to $Y.rd = \rho \geq r$, and let us prove it for $\rho+1$. To this end, we have to show that $Y.val = X.val$ for every Y that is written in REG with $Y.rd = \rho+1$. Let us assume by contradiction that $Y.val \neq X.val$ and let p_i be the first process that writes $\langle \rho+1, -, -, Y.val\rangle$ into REG. This happens at line 04 or 05. In all cases, this implies that, at this moment, $view_j$ is $\mathcal{H}(\langle \rho, -, -, Y.val\rangle)$. But, according to the induction assumption, this implies $Y.val = X.val$, a contradiction which completes the proof of the lemma. $\square_{Lemma\ 10}$

4.5 Proof of the algorithm: exploiting the previous lemmas

Lemma 11 *No two processes decide different values.*

Proof Let r be the smallest round in which a process decides, p_i and val being the deciding process and the decided value, respectively. Therefore, there is a time τ at which $view_i^\tau$ is $\mathcal{H}(\langle r, \mathtt{up}, \mathtt{false}, val\rangle)$. Due to Lemma 10, every homogeneous snapshot starting from round r is necessarily associated with the value val. 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. $\square_{Lemma\ 11}$

Lemma 12 *For every quadruplet X that is written in REG, $X.val$ is a value proposed by some process.*

Proof Let us assume by contradiction that $X.val = v$ was not proposed by a process, and let p_i be the first process that writes X into REG. We consider two cases according to the line at which the write occurs.

- v is written into REG at line 04 or line 05. In this case, p_i obtained a view of REG in which at least some register contains the value v. According to the predicate of these two lines, the round number associated with v is necessarily greater than 0 which implies that v was previously written into REG and was not there initially. But this means that p_i is not the first process which writes v into REG, a contradiction.
- v is written into REG at line 08. In this case, the quadruplet X, where $X.val = v$, was returned by the call of the function $\mathsf{sup}()$, namely $\mathsf{sup}(view[1], \cdots, view[n], \langle 1, \mathtt{down}, \mathtt{false}, v_i\rangle)$, from which it follows that v is either v_i (the proposal of p_i) or some value that was previously written by another process. But, by assumption, p_i is assumed to be the first process to write v. Hence, $v = v_i$, which concludes the proof of the lemma. $\square_{Lemma\ 12}$

Lemma 13 *A decided value is a proposed value.*

Proof If a process decides a value v, it does it at line 03. Hence, according to the predicate of line 03, the round number associated with this value is greater than 0 which means that v was necessarily written into REG by some process. It then follows from Lemma 12, that v was proposed by a process, which establishes the claim. $\square_{Lemma\ 13}$

Lemma 14 *Let T be a set of quadruplets. For every $T' \subseteq T$: $\mathsf{sup}(T' \cup \{\mathsf{sup}(T)\}) = \mathsf{sup}(T)$.*

Proof Let $S = \mathsf{sup}(T)$. Hence $S.rd$ is the highest round number in T. Moreover, S is greater than, or equal to, any quadruplet in T. Hence, $\mathsf{max}(T' \cup \{S\}) = S$. Therefore, combined with the the definition of $\mathsf{sup}()$, we have: $\mathsf{sup}(T' \cup \{S\}) = \langle S.rd, S.\ell v\ell, conflict(T' \cup \{S\}), S.val\rangle$. Thus, in order to prove that $\mathsf{sup}(T' \cup \{S\}) = S$, we need to show that $conflict(T' \cup \{S\}) = S.cf\ell$. There are two cases depending on the value of $S.cf\ell$.

- $S.cf\ell = \mathtt{true}$.
  In this case, $conflict1(\{S\}) = \mathtt{true}$. But $S.rd$ is the highest round number in T from which it follows that $S.rd$ is also the highest in $T' \cup \{S\}$. Therefore, $conflict1(\{S\}) = \mathtt{true}$ implies that $conflict1(T' \cup \{S\}) = \mathtt{true}$.
- $S.cf\ell = \mathtt{false}$.
  Since $S = \mathsf{sup}(T)$, it follows that $conflict(T) = \mathtt{false}$. Consequently, both $conflict1(T)$ and $conflict2(T)$ are $\mathtt{false}$. Moreover, as $S.cf\ell = \mathtt{false}$, it follows that $conflict1(\{S\}) = \mathtt{false}$. Therefore $conflict1(T \cup \{S\}) = \mathtt{false}$. But, as $T' \subseteq T$, this yields $conflict1(T' \cup \{S\}) = \mathtt{false}$.

  On another side, it follows from $conflict2(T) = \mathtt{false}$ that $|vals(T)| = 1$. As $S = \mathsf{sup}(T)$, we have $S.val \in vals(T)$. Therefore $|vals(T \cup \{S\})| = 1$. Since $T' \subseteq T$, it follows that $|vals(T' \cup \{S\})| = 1$ which implies $conflict2(T' \cup \{S\}) = \mathtt{false}$.

  As both $conflict1(T' \cup \{S\})$ and $conflict2(T' \cup \{S\})$ are false, it follows that $conflict(T' \cup \{S\}) = \mathtt{false}$.

From the case analysis we conclude that $conflict(T' \cup \{S\}) = S.cf\ell$. $\square_{Lemma\ 14}$

Lemma 15 *If there is a time after which a process executes solo, it decides a value.*

Proof Assume that p_i eventually runs solo, we need to show that p_i decides. There exists a time τ, after which no other process than p_i writes into REG. Let $\tau' \geq \tau$ be the first time at which p_i takes a snapshot after τ. This snapshot is well defined, as p_i runs solo after τ and the implementation of atomic snapshot is obstruction-free. Let $S = \mathtt{sup}(view_i^{\tau'}[1], \cdots, view_i^{\tau'}[n], \langle 1, \mathtt{down}, \mathtt{false}, v_i\rangle)$.

Let us first show that there is a time after τ at which REG is $\mathcal{H}(S)$.

- If $REG^{\tau'}$ is $\mathcal{H}(S)$, we are done.
- If $REG^{\tau'}$ is not $\mathcal{H}(S)$, p_i executes line 06 and computes S. Then it writes S in an entry of REG (containing a value different from S), and re-enters the loop. If REG is then $\mathcal{H}(S)$, we are done. Otherwise, p_i executes again line 06 and, due to Lemma 14, the quadruplet computed by the function $\mathtt{sup}()$ is equal to S. It follows that after a finite number of iterations of the loop, REG is $\mathcal{H}(S)$.

When REG is $\mathcal{H}(S)$, we have the following.

- If $S = \langle -, \mathtt{up}, \mathtt{false}, -\rangle$, p_i decides in line 03.
- If $S = \langle r, \mathtt{down}, \mathtt{false}, val\rangle$, then p_i writes $Y = \langle r+1, \mathtt{up}, \mathtt{false}, val\rangle$ in line 04. Using the same argument as above, there is a time at which REG becomes $\mathcal{H}(Y)$, and the previous case holds.
- If $S = \langle r, -, \mathtt{true}, val\rangle$, then p_i writes $Y = \langle r+1, \mathtt{down}, \mathtt{false}, val\rangle$ in line 05. Then p_i keeps writing Y in the following iterations until REG becomes $\mathcal{H}(Y)$, and the previous case holds.

Hence, in all cases p_i eventually decides. $\square_{Lemma\ 15}$

Lemma 16 *If a single value is proposed, all correct processes decide.*

Proof Let us assume that all processes propose the same value v. It follows that all the processes keep writing $X = \langle 1, \mathtt{down}, \mathtt{false}, v\rangle$ until REG becomes $\mathcal{H}(X)$. Then, once every register of REG has been updated at least once, the processes start writing $Y = \langle 2, \mathtt{up}, \mathtt{false}, v\rangle$ until REG becomes $\mathcal{H}(Y)$ and v. When this occurs, v is decided. $\square_{Lemma\ 16}$

Theorem 1 *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 11 (Agreement), Lemma 13 (Validity), Lemma 15 (OB-Termination), and Lemma 16 (SV-Termination). $\square_{Theorem\ 1}$

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

The algorithm The 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 the memory REG.

As far as the k-set agreement property is concerned (no more than k different values can be 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 ($\mathtt{return}()$ at line 04). 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 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, the 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 $sn_i = 1,\ 2,\ \ldots$ is the sequence number of its current instance, and v_i 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 that, it is possible to solve the repeated problem with only $(n - k + 1)$ atomic registers. According to the complexity results of [9], it follows that this algorithm is optimal in the number of atomic registers, which consequently closes the lower/upper bounds discussion associated with 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 new field $X.sn$ contains a sequence number, while the new 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 its first five fields while the relation "$\sqsupset$" is now defined as follows:

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

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

- The variable sn_i, initialized to 0, is used by p_i to generate its sequence numbers. It is assumed that p_i increases 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. The parts which are new with respect to the base algorithm of Figure 2 are in red.

```
operation propose(sn_i, v_i) is
(01) repeat forever
(02)   view ← REG.snapshot();
(03)   case (∀x : view[x] = ⟨sn_i, r, up, false, vaℓ, −⟩   where r > 0) then dcd_i[sn_i] ← vaℓ; return(vaℓ)
(04)        (∀x : view[x] = ⟨sn_i, r, down, false, vaℓ, −⟩ where r > 0) then REG.write(1, ⟨sn_i, r + 1, up, false, vaℓ, dcd_i⟩)
(05)        (∀x : view[x] = ⟨sn_i, r, ℓevel, true, vaℓ, −⟩ where r > 0) REG.write(1, ⟨sn_i, r + 1, down, false, vaℓ, dcd_i⟩)
(06)        otherwise let ⟨inst, r, ℓevel, conflict, vaℓ, dec⟩ ← sup(view[1], ··· , view[n], ⟨sn_i, 1, down, false, v_i, dcd_i⟩);
(07)                  if (inst > sn_i) then dcd_i[sn_i] ← dec[sn_i]; return dcd_i[sn_i] end if
(08)                  x ← smallest index such that view[x] = min(view[1], ··· , view[n]);
(09)                  REG.write(x, ⟨inst, r, ℓevel, conflict, vaℓ, dec⟩)
(10)   end case
(11) end repeat.
```

Figure 3: Repeated obstruction-free Consensus

- Line 03. When all entries of a view obtained by p_i contain only sixuplets whose the first five fields are equal, p_i decide the value $va\ell$. But before returning $va\ell$, p_i writes it in $dcd_i[sn_i]$. Hence, when p_i will execute the next (n, k)-set agreement instance (whose occurrence number will be $sn_i + 1$), it will be able to help processes, whose current sequence number sn' are smaller than sn_i, decide a value returned by the instance sn' of the repeated (n, k)-set agreement.
- Line 04. In this case, p_i obtains a view whose five first entries are equal to $\langle sn_i, r, \texttt{down}, \texttt{false}, va\ell\rangle$. It then writes in $REG[1]$ the sixuplet $\langle sn_i, r, \texttt{down}, \texttt{false}, va\ell, dcd_i\rangle$. Let us notice that the write of dcd_i is to help other processes decides in (n, k)-set agreement instances whose sequence number is smaller than sn_i.
- Line 05. This case is similar to the previous one.

- Lines 06-10. In this case, p_i computes the supremum of the snapshot value $view$ obtained at line 03 plus the qsixuplet $\langle sn_i, 1, \mathtt{down}, \mathtt{false}, val\ell, dcd_i\rangle$. There are two cases.
  - 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 smaller than $inst$. 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 $inst = sn_i$, p_i executes as in the base algorithm (lines 08-09).

Hence, 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 already indicated, it follows from the lower bound established in [9] that this algorithm is optimal with respect to the number of underlying atomic registers.

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. Let $x \leq k$ ([6]).

One-shot x-obstruction-freedom This progress condition, introduced in [30, 31], is a natural generalization of obstruction-freedom, which corresponds to the case $x = 1$.

x-Obstruction-freedom guarantees that, for every set of processes P, $|P| \leq x$, every correct process in P returns from its operation invocation if no process outside P takes steps for "long enough". 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.

x-Obstruction-free (n, k)-set agreement: OB-Termination When considering 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 only property that must be adapted is OB-Termination, which 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 REG To cope with the x-concurrency allowed by obstruction-freedom, the array REG is such that it has now $m = n - k + x$ entries (i.e., $m = n - k + 1$) entries for the base obstruction-freedom). This increase in the size of the array is due to the fact that the algorithm is required to terminate in more scenarios than simple 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 made up of a single $X.val$, by a set of values denoted $X.valset$.

function $\mathsf{sup}(T)$ **is** % S is a set of quadruplets, the last field of each of them is now a set of values %
(S1') **let** $\langle r, \ell evel, cf\ell, valset\rangle$ **be** $\mathsf{max}(T)$; % lexicographical order %
(S2') **let** $vals(T)$ **be** $\{v \mid \langle r, -, valset\rangle \in T \ \wedge\ v \in valset\}$;
(S3) **let** $conflict1(T)$ **be** $\exists\, \langle r, -, conflict, -\rangle \in T$; % conflict inherited %
(S4') **let** $conflict2(T)$ **be** $|vals(T)| > x$; % conflict discovered %
(S5) **let** $conflict(T)$ **be** $conflict1(T) \vee conflict2(T)$;
(N) **if** $conflict(T)$ **then** $vals'(T) \leftarrow valset$ **else** $vals'(T) \leftarrow$ the set of the (at most) x greatest values in $vals(T)$ **end if**;
(S6') $\mathsf{return}\big(\langle r, \ell evel, conflict(T), vals'(T)\rangle\big)$.

Figure 4: Function $\mathsf{sup}()$ suited to x-obstruction-freedom

[6] This assumption is a necessary requirement to solve (n, k)-set agreement in a read/write system. It 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].

The modified function sup() Coping with x-concurrency requires to also adapt the function sup(). This function sup() is a simple extension of the base version described in Figure 1, that allows to consider a set of values instead of a single value. It is described in Figure 4. The lines that are modified (with respect to the base function sup()) are followed by a "prime", and a new line (marked N) is added. More precisely, the modifications are the following.

- Line S1'. The last field of a quadruplet is now a set of values, denoted $valset$. As far as the lexicographical ordering is concerned, the sets $valset$ are ordered as follows. They are ordered by size, and 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: not modified.
- 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 $conflict(T) = \texttt{true}$. Otherwise, it contains the (at most) x greatest values of $vals(T)$.
- Line S6'. The quadruplet returned by $\mathsf{sup}(T)$ differs from the one of Figure 2 in its last field which is now the set $vals'(T)$.

It is easy to see that, when the last field of the quadruplets is reduced to singleton, and $x = 1$, this extended version boils down to the one described in Figure 2.

```
operation propose(v_i) is
(01) Q ← ⟨1, down, false, {v_i}⟩;
(02) repeat forever
(03)    view ← REG.snapshot();
(04)    case (∀x : view[x] = Q = ⟨r, up, false, valset⟩     where r > 0) then return any value in valset
(05)         (∀x : view[x] = Q = ⟨r, down, false, valset⟩   where r > 0) then Q ← ⟨r + 1, up, false, valset⟩; REG.write(1, Q)
(06)         (∀x : view[x] = Q = ⟨r, levℓ, true, valset⟩    where r > 0) then let v be any value in valset;
(07)                                                                      Q ← ⟨r + 1, down, false, {v}⟩; REG.write(1, Q);
(08)         otherwise let Q ← sup(view[1], ··· , view[n], Q);
(09)                       x ← smallest index such that view[x] ≠ Q;
(10)                       REG.write(x, Q)
(11)    end case
(12) end repeat.
```

Figure 5: Anonymous x-obstruction-free Consensus

x-Obstruction-free (n, k)-set agreement: algorithm An algorithm extending the base obstruction-free algorithm of Figure 2 to an x-obstruction-free (n, k)-set agreement algorithm is described in Figure 2. (Let us remember that, as the underlying snapshot algorithm is non-blocking [15], it ensures that –whatever the concurrency pattern– at least one snapshot invocation always terminates.) This algorithm solving the x-obstruction-free (n, k)-set agreement problem is obtained as follows, where (as already indicated) the array REG is composed of $m = n - k + x$ atomic read/write registers.

- 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:

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

- Each process p_i maintains a local quadruplet denoted Q, containing the last quadruplet it has computed. Initially, Q is equal to $\langle 1, \texttt{down}, \texttt{false}, \{v_i\}\rangle$ (line 01)[7].

  This quadruplet allows its owner p_i to have an order on the all the quadruplets it champions during the execution of $\mathsf{propose}(v_i)$. Hence, 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 the x-OB-termination property.

  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 $\langle r, down, \texttt{false}, valset\rangle$, or $\langle r, \ell evel, \texttt{true}, valset\rangle$. Hence, according to the terminology of the proof of the base algorithm, introduced in Section 4.1, $view$ is homogeneous, i.e., $view$ is $\mathcal{H}(Q)$ where Q obeys some predefined pattern.

[7] Let us notice that, the algorithm has no longer the *memoryless* property of the base algorithm.

- Lemma 10 needs to be re-formulated to take into account the set field of each 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 if the number of participating processes does not exceed the number of available registers in REG.
- As far the k-set agreement property (no more than k different values can be decided), we have to show that $(n-k+x)$ registers are sufficient. The reasoning is similar to one done at the end of 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 02) 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 of line 03, $REG^{\tau'}$ is $\mathcal{H}(X)$ for some X with $X.\ell v\ell = \texttt{up}$ and $X.cf\ell = \texttt{false}$. Observe that $|X.valset| \leq x$.

  According to the new statement of Lemma 10, since starting from τ the number of participating processes is always less than the number of registers, then all deciding snapshots 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'.
- As far as x-OB-termination is concerned, the key is line 07. When a process p_i detects a conflict ($Q.cf\ell = \texttt{true}$, at line 06), it starts a new round with a set which is a singleton. Hence, if there is a finite time after which no more than x processes are executing, there is a finite round from which at most x values survive and appear in the next round. From that round, no new conflict can be discovered, and eventually the (at most) x running processes obtain snapshots entailing decision.

8 Conclusion

This paper presented first a base a one-shot obstruction-free (n, k)-set agreement algorithm for a system made up of n asynchronous and anonymous processes, which communicate through atomic read/write registers. This algorithm requires only $(n - k + 1)$ such registers. From this cost point of view, it is the best algorithm known so far (the best previously known algorithm requires $2(n - k) + 1$ atomic read/write registers). Hence, this algorithm answers the challenge posed in [7], and establishes a new upper bound of $(n - k + 1)$ on the number of registers to solve the one-shot obstruction-free (n, k)-set agreement problem. This upper bound improves the ones stated in [9] for anonymous and non-anonymous systems.

A simple extension of the previous algorithm has then been presented, that solves the repeated (n, k)-set agreement problem. While the lower bound of $(n-k+1)$ atomic registers was established in [9] for this problem, the proposed algorithm shows that the upper bound is also equal $(n-k+1)$, and consequently the proposed algorithm is optimal. The paper has also generalized the base one-shot algorithm to solve the (n, k)-set agreement problem in the context of x-obstruction-freedom. The corresponding algorithm reduces to $(n - k + x)$ the upper bound on the number of atomic read/write registers.

To attain these goals the algorithms, which have been presented in an incremental way, rely on a simple round-based structure. Moreover, the base one-shot algorithm does not require persistent local variables, and, in addition to a proposed value, an atomic register contains only two bits and a round number. The algorithm solving the repeated (n, k)-set agreement problem requires that each atomic register includes two more integers.

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. The paper has shown 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 $1 \leq x \leq k < n$.

Acknowledgments

This work has been partially supported by the French ANR project DISPLEXITY devoted to computability and complexity in distributed computing, and the Franco-German ANR project DISCMAT devoted to connections between mathematics and distributed computing.

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

[6] 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)

[7] 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)

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

[9] 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)

[10] Ellen Fich F., Herlihy M., and Shavit N., On the space complexity of randomized synchronization. *Journal of the ACM*, 45(5):843-862 (1998)

[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] Gafni E. and Lamport L., Disk Paxos. *Distributed Computing*, 16(1):1-20 (2003)

[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] Taubenfeld G., On the Computational Power of Shared Objects. *Proc. 13th Int'l Conference On Principle Of Distributed Systems (OPODIS 2009)*, Springer LNCS 5923, pp. 270-284 (2009)

A Non-blocking snapshot object

This appendix presents a non-blocking (hence obstruction-free) snapshot object which uses no additional atomic register. The idea that underlies this algorithm, which is due to Guerraoui and Ruppert [15], is simple. The algorithm, described in Figure 6, considers that the n anonymous processes share m underlying MWMR atomic registers.

Shared variables
$SM[1..m]$: array of n multivalued MWMR atomic registers, initially $[\langle -, \perp\rangle, \cdots, \langle -, \perp\rangle]$;
$SM[x] = \langle SM[x].ts, SM[x].value\rangle$; only $SM[i].value$ can be made visible outside.

Permanent local variable: each process $p_i$manages a counter ts_i, initialized to 0.

operation write(x, v) **is** % issued by p_i %
(01) $SM[x] \leftarrow \langle ts_i, v\rangle$; $ts_i \leftarrow ts_i + 1$; return().

operation snapshot() **is**
(02) $count \leftarrow 1$; **for each** $x \in \{1, \ldots, m\}$ **do** $sm1[x] \leftarrow SM[x]$ **end for**;
(03) **repeat forever**
(04) **for each** $y \in \{1, \ldots, m\}$ **do** $sm2[y] \leftarrow SM[y]$ **end for**;
(05) **if** $(\forall\, x \in \{1, \cdots, m\} :\ sm1[x] = sm2[x])$
(06) **then** $count \leftarrow count + 1$;
(07) **if** $(count = m(n-1)+2)$ **then** return$(sm1[1..m].value)$ **end if**
(08) **else** $count \leftarrow 1$
(09) **end if**;
(10) $sm1[1..m] \leftarrow sm2[1..m]$
(11) **end repeat**.

Figure 6: Obstruction-free snapshot object [15]

Each process p_i manages an integer local variable ts_i, that it uses to associate a sequence number to its successive write operations into any atomic register $SM[x]$ (line 6).

When a process invokes snapshot(), it repeatedly reads the array $SM[1..m]$ until it obtains an array value $sm[1..m]$ that does not change during $(m(n-1)+2)$ readings of $SM[1..m]$. When this occurs, the invoking process returns the corresponding array value $sm[1..m]$.

Trivially, any write operation terminates. As far the snapshot operation is concerned, it is easy to see that, if there is a time after which a process executes alone it terminates its snapshot operation, hence the implementation is obstruction-free.

To show that it is non-blocking, let us assume that a process invokes repeatedly $REG[x]$.write() (whatever x) followed by REG.snapshot() (as it is the case in the algorithms presented in the paper). An invocation of REG.snapshot() can be prevented from terminating only if processes issue permanently invocations of write(), Let us assume that no invocation of REG.snapshot() terminates. This means that there are processes that permanently issue write operations. But this contradicts the assumption that each processes alternates invocations of $REG[x]$.write() (whatever x) and REG.snapshot(). This is because, between two writes issued by a same process, this process invoked REG.snapshot(), and consequently this snapshot invocation terminated.

As far the linearization of the operations write() and snapshot() invoked by the processes is concerned we have the following (this proof is from [15]). Let us consider an invocation of snapshot() that terminates. It has seen $m(n-1)+2$ times the same vector $sm[1..m]$ in the array $SM[1..m]$. Since a given pair $\langle ts, v\rangle$ can be written at most once by a process, it can be written at most $(n-1)$ times during a snapshot (once by each process, except the one invoking the snapshot). It follows that, among the $m(n-1)+2$ times where the same vector $sm[1..m]$ was read from $SM[1..m]$, there are least two consecutive reads during which no process wrote a register. The snapshot invocation is consequently linearized after the first of these two reads.

B All Correct Processes Decide if One Process Decides

This appendix shows that, by adding one MWMR register, the consensus termination property can be strengthened. More precisely, we have then the additional termination property (where OA stands for "One-All").

- OA-termination. If a process decides, all correct processes decide.

Let DEC be the additional register, initialized to the default value $\perp$. The extended algorithm is the one described in Figure 2 with only two modifications.

- The first modification is the addition of the new line

if $(DEC \neq \perp)$ **then** return(val) **end if**

between line 01 and line 02. Each time it enters the repeat loop, a process first checks if a value was previously decided. If it is the case, it decides it.

- The first modification is the addition, at line 04, of the statement "$DEC \leftarrow val$", just before the statement "return(val). When a process is about to decide, it first writes the decided value in the MWMR atomic register DEC.

Theorem 2 *The extended algorithm solves the obstruction-free consensus problem satisfying the additional* OA*-termination property, with* $(n+1)$ *underlying* MWMR *atomic registers.*

Proof The proof follows directly from the proof of the base algorithm of Figure 2 (OB-termination and SV-termination) and the fact that no process can block while executing the repeat loop (hence OB-termination $\Rightarrow$ OA-termination). $\square_{Theorem\ 2}$