Noname manuscript No.
(will be inserted by the editor)

The k-simultaneous consensus problem

Y. Afek · E. Gafni · S. Rajsbaum · M. Raynal · C. Travers

the date of receipt and acceptance should be inserted later

Abstract This paper introduces and investigates the k-simultaneous consensus task: each process participates at the same time in k independent consensus instances until it decides in any one of them. It is shown that the k-simultaneous consensus task is equivalent to the k-set agreement task in the wait-free read/write shared memory model, and furthermore k-simultaneous consensus possesses properties that k-set does not. In particular we show that the multivalued version and the binary version of the k-simultaneous consensus task are wait-free equivalent. These equivalences are independent of the number of processes. Interestingly, this provides us with a new characterization of the k-set agreement task that is based on the fundamental binary consensus problem.

Keywords: Asynchronous shared memory systems, Binary vs multivalued agreement, Consensus, Distributed computability, Process crash, Set agreement, Wait-free construction.

A preliminary draft of this paper has been presented at the conference ICDCN'06 [3].

Partially supported by PAPIIT-UNAM project IN116808.

Partially supported by the French ANR project SHAMAN.

Y. Afek
Computer Science Department, Tel-Aviv University, Israel 69978, E-mail: afek@cs.tau.ac.il

E. Gafni
Department of Computer Science, UCLA, Los Angeles, CA 90095, USA, E-mail: eli@cs.ucla.edu

S. Rajsbaum
Instituto de Matemáticas, UNAM, D. F. 04510, Mexico E-mail: rajsbaum@math.unam.mx

M. Raynal
Université de Rennes 1, IRISA, Campus de Beaulieu, 35042 Rennes, France E-mail: raynal@irisa.fr

C. Travers
Department of Computer Science, Technion, Haifa 32000, Israel E-mail: corentin@cs.technion.ac.il

1 Introduction

Context and motivation of the paper In the consensus task, each process proposes a value, and it is required that (1) each non-faulty process decides on a value (*termination*) in such a way that (2) there is a single decided value (*agreement*), and (3) the decided value is one of the proposed values (*validity*). Unfortunately, this problem has no solution in asynchronous systems as soon as even only one process may crash, be the system a shared memory system [18] or a message passing system [10].

One way to weaken the consensus problem is to allow several different values to be decided. This approach has given rise to the *k-set agreement* problem where up to k different values can be decided [7]. While this problem (sometimes also called k-set consensus) can be solved despite asynchrony and process failures when $k > t$ (where t is the maximum number of processes that can be faulty), it has been shown that it has no solution when $t \geq k$ [6, 15, 22].

This paper presents and investigates another way to weaken the consensus problem. The intuition that underlies this problem, called here *scalar k-simultaneous consensus*, is "win one out of several". More explicitly, each process proposes a value in k independent consensus instances, the same value to all instances. It is required that every correct process decides on a value in at least one consensus instance. In other words, a process decides on at least one pair composed of a value and a consensus instance number. Two processes can decide on different pairs; however if they decide

on the same consensus instance they also decide on the same value (that has been proposed by one of the processes)[1]. We also consider an equivalent *vector* version of the k-*simultaneous consensus*, where each process proposes k possibly different values, one value to each of the k independent consensus instances. Again a process decides on a pair composed of a value and a consensus instance number. Two processes can decide on two different pairs; if they decide on the same consensus instance they also decide on the same value (that has been proposed to that instance). It is easy to see that the scalar version and the vector version of the k-simultaneous consensus task are equivalent (see Section 2.4).

As explained in [13], simultaneous consensus can be useful in situations where several processes participate concurrently in k different applications: a k-simultaneous consensus solution can guarantee wait-free progress in at least one application. Indeed, recently this problem has been instrumental in determining the weakest failure detector that wait-free solves the $(N-1)$-set agreement problem in asynchronous read/write shared memory systems made up of N processes [23]. In addition to its possible applications, a simple and natural generalization of the simultaneous consensus is the simultaneous set-consensus, i.e., the case where each of the k consensus instances is replaced by an instance of another agreement task (this point is investigated in the conclusions section where each consensus instance is replaced by an ℓ-set agreement instance).

In this paper we address two questions, (see Figure 1) the first question addresses the relation between the k-set agreement problem and the k-simultaneous consensus problem. While, given a solution to the k-simultaneous consensus problem, it is easy to solve the k-set agreement problem, what about the other direction? In other words, are these problems equivalent? We answer this question positively by presenting a wait-free transformation that, given a k-set agreement task, builds a k-simultaneous consensus task.

The second question addressed in this paper concerns the relation between the multivalued and the binary version of the k-simultaneous consensus. In the binary version each consensus instance is a binary consensus: each process proposes either 0 or 1 to each consensus instance. We consider only the vector version of the problem. Indeed for $k \geq 2$, the scalar version is trivial, since each binary input value can be deterministically assigned to a consensus instance. While it is known that the multivalued consensus and the binary consensus are equivalent (e.g., [21]), the same equivalence cannot be achieved in the k-set consensus realm, since it is meaningless to talk about binary k-set consensus. What about k-simultaneous multivalued consensus and k-simultaneous binary consensus? The binary version of the problem is a simple case of the multivalued one, but what about the other direction? It is shown in this paper that the two problems are equivalent by presenting a wait-free transformation that, given k-simultaneous binary consensus tasks, builds a k-simultaneous multivalued task.

Hence, the paper shows that the k-set agreement problem and the k-simultaneous binary consensus problem are equivalent. Intuitively, this means that, given a solution to any one of these problems, it is possible to wait-free solve the other one in an asynchronous read/write shared memory system prone to any number of process crashes. Thus, while, unlike consensus, k-set agreement has no binary version, the previous equivalence provides a characterization of k-set agreement in terms of k simultaneous instances of the binary consensus problem. This is summarized in Figure 1.

Roadmap Section 2 describes the computation model and presents the problems we are interested in. Section 3 shows that the k-set agreement problem and the k-simultaneous consensus problem are equivalent. Section 4 shows that the k-simultaneous multivalued consensus problem is not more powerful than its binary counterpart. Finally, in Section 5 the conclusions are provided.

2 Computation model and problem definitions

2.1 Computation model

Processes The system consists of an arbitrary number of processes denoted $p_i, p_j, \ldots$ The integer i is the identity of p_i, and no two processes have the same identity. A run is a sequence of steps of a number of processes with unique identity. The processes that appear in a run are called *participating* processes. An infinite number of processes may participate in an infinite run and the number of active processes simultaneously may grow without bounds. This is the infinite arrival model with unbounded concurrency, introduced and investigated in [11,20]. A process that participates in a run is provided with a local constant whose value is its identity. It is not provided with local variables

[1] Let us notice that the words "simultaneous consensus" have been used with a different meaning in round-based synchronous systems. In these systems, they mean that all the processes that participate in a consensus instance have to terminate during the very same round [8,9].

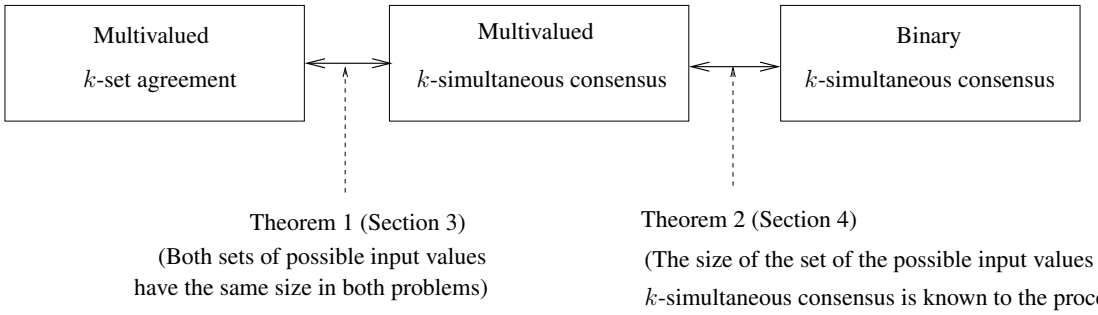

Fig. 1 Equivalences among the problems

whose values would allow it to compute the number of participating processes or their identities.

Processes are asynchronous, there is no assumption on their relative speeds. Moreover, any number of processes may crash. Before it crashes (if it ever crashes), a process executes correctly its algorithm. A crash is a premature halt: after it has crashed, a process executes no more operations. Given a run, a process that does not crash is *correct* in that run, otherwise it is *faulty* in that run.

A remark on the number of processes: Most distributed algorithms are designed for a set of N processes where N is fixed and known by every process. Moreover, each process is assigned a unique identity comprised between 1 and N, and an algorithm can make use of both the number of processes and their identity.

In contrast, the algorithms designed in this paper work with an arbitrary number of processes. Such a situation occurs in systems that dynamically change over time. For example, a network may allow nodes to be added or removed, or an operating system may allow processes to dynamically join, participate in a distributed algorithm and finally leave. Algorithms for infinitely many processes (e.g. [11,20]) have recently received attention. Their advantages over algorithms for a fixed number of processes are significant [4]: (1) They have no system size parameters to configure, and (as a result) they are more robust and elegant; (2) They automatically handle the crash/recovery of processes (as a process that crashes and recovers can join the algorithm simply by assuming a new identity); (3) They guarantee progress even if processes keep on arriving (which is important in loosely-coupled systems, like peer-to-peer systems, where there is a large number of nodes that come and go all the time).

Communication model The processes communicate by way of reliable multi-reader/multi-writer atomic registers [5,17,19]. In addition the algorithms presented here use the atomic snapshot primitive [1]. This basic operation, denoted $SM.\mathsf{set\_snapshot}()$ where SM is a shared array with one entry per process, returns a set of values that were simultaneously present in SM during the snapshot operation. Such a set is also called a *snapshot* in the sequel. Any set of read and write operations on individual cells of the array SM, and $SM.\mathsf{set\_snapshot}()$ operations, is linearizable [16]. Therefore, if each cell in the array is written only once and no value can ever be removed, the sets obtained by a sequence of $SM.\mathsf{set\_snapshot}()$ operations are such that each set contains all the ones that precede it in sequence (also called linearization order). We say that this sequence of sets satisfies the *containment* property. A wait-free snapshot algorithm for the infinite arrival model with unbounded concurrency is described in [11].

Remark All the algorithms described in the paper are given for an arbitrary process p_i. Uppercase letters are used to denote shared tasks or objects, while lowercase letters are used for local variables (these variables are subscribed with the index of the corresponding process).

2.2 Problem definitions

Decision problems To model decision problems, we identify two special local variables in each process p_i: an input variable denoted $input_i$ and an output variable denoted dec_i. The local variable $input_i$ is initialized with some value v drawn from a set $\mathcal{I}$ of possible input values. We say that "process p_i *proposes* the value v" when v is the value in its $input_i$ variable when it wakes up. The local variable dec_i, initialized to $\perp$, can be written only once. When it takes a value w different from $\perp$, we say that "process p_i *decides* on the value w". The value $\perp$ is a default value not in $\mathcal{I}$.

A task T is a one-shot decision problem specified by a set of input values $\mathcal{I}$, a set of output values $\mathcal{O}$ and a relation that specifies, for each assignment of values

in $\mathcal{I}$ to the processes, which output values each process is allowed to decide on.

In the tasks investigated in this paper, the set $\mathcal{I}$ of input values is totally ordered, and n denotes the number of elements in $\mathcal{I}$. We assume that $k < n$, where k is the central parameter used in the specification of the decision problems investigated in this paper (k-set-agreement and k-simultaneous consensus).

An algorithm A (we also say an "object") solves a task T if:

- A provides each process with a single operation denoted $A.\mathsf{propose}()$. That operation takes as input any value in $\mathcal{I}$ and returns values in $\mathcal{O}$, where $\mathcal{I}$ and $\mathcal{O}$ are the input and output sets associated with T (see above).
- In any execution in which $A.\mathsf{propose}()$ is invoked at most once by each process, the values returned by any $A.\mathsf{propose}()$ invocation complies with the specification of T.

The k-set agreement problem As indicated in the Introduction, the k-set agreement problem [7] is a generalization of the consensus problem (that corresponds to the case $k = 1$). It is defined by the following properties.

- Termination: each correct process decides on a value.
- Validity: a decided value is a proposed value.
- Agreement: at most k different values are decided.

As for all the problems considered in this paper, the termination property requires a solution based on wait-free algorithms [14]: a correct process has to terminate regardless of the number of faulty processes.

The k-set agreement problem could be defined for a *binary* input set, by restricting the set of input values $\mathcal{I}$ to the set $\{0, 1\}$. However, while there is no wait-free solution to the binary consensus problem, the binary k-set agreement problem can be trivially solved when $k > 1$.

Let KSA be an object that solves the k-set agreement problem. It provides the processes with a single operation denoted $KSA.\mathsf{set\_propose}_k()$. That operation takes a proposed value as input parameter, and returns a decision value.

The k-simultaneous consensus problem [2] Both (the scalar and the vector) versions of the k-simultaneous consensus problem consist of k independent instances of the consensus problem where a process is required to decide in at least one of them. More precisely, in the *scalar version*, process p_i proposes the same value v_i to each of the consensus instances. In the *vector version*, process p_i proposes a vector $[v_i^1, \cdots, v_i^k]$ where v_i^e is the value it proposes to the e-th consensus instance ($1 \leq e \leq k$). Each process decides on pairs $\langle c, d\rangle$ where c is a consensus instance and d is a value. The problem is defined by the following properties.

- Termination: each correct process decides on at least one pair.
- Validity: if a process p_i decides $\langle c, d\rangle$, then c is a consensus instance (i.e., $1 \leq c \leq k$), and d is a value that has been proposed to that consensus instance.
- Agreement: if the pairs $\langle c, d\rangle$ and $\langle c, d'\rangle$ are decided, then $d = d'$.

Similarly to the k-set-agreement problem, we define the binary k-simultaneous consensus problem by restricting the set of input values $\mathcal{I}$. More precisely, we have $\mathcal{I} = \{0, 1\}$ for the scalar binary k-simultaneous consensus problem (each process proposes 0 or 1), and $\mathcal{I} = \{0, 1\}^k$ for its vector version (each process proposes a size k vector made up of 0's and 1's). As for the k-set agreement problem, it is easy to see that the scalar version of the binary k-simultaneous consensus problem can trivially be solved when $k > 1$.

It is important to remark that, for $k = 1$, both the scalar and the vector version of the binary simultaneous consensus problem boil down to the binary consensus problem. In the remainder of the paper, we use explicitly the word "binary" when we discuss the binary version of a problem. When we discuss their non-binary versions, we sometimes use the word "multivalued".

Let KSC be an object that solves the k-simultaneous consensus problem. It provides the processes with a single operation denoted $KSC.\mathsf{sc\_propose}_k()$. In the scalar version, that operation takes as input parameter the process input value, and in the vector version it takes a vector with k proposed values (one for each consensus instance). That operation returns a pair $\langle c, d\rangle$. In the case of a binary k-simultaneous consensus object, the operation is denoted $\mathsf{bin\_sc\_propose}_k()$.

2.3 Problems equivalence

For comparing decision problems (tasks) we use wait-free constructions. Namely, for two problems (tasks) $P1$ and $P2$, we say that "$P1$ *solves* $P2$" if there is a wait-free algorithm $\mathcal{A}$ that solves $P2$ using any number of copies of objects that solve $P1$ (in addition to any number of read/write atomic registers). If $P1$ and

[2] This problem originates from our previous research where we introduced and investigated the *musical benches* problem [12], and the *committee decision* problem [13]. The k-simultaneous consensus problem generalizes both of them.

$P2$ solve each other, the problems are said to be *equivalent*.

All the constructions described in the paper are wait-free and work for an arbitrary number of processes. Moreover, when we compare the multivalued versions of the problems defined above, we assume that the size of the set of input values in both problems is the same, namely n.

2.4 The multivalued scalar version and vector version are equivalent

It is easy to see that the vector version and the scalar version are equivalent (i.e., each one can implement the other one) when the size of the set $\mathcal{I}$ of input values is the same in both problems.

From the vector version to the scalar version Implementing the scalar version from the vector version is trivial. Let v_i be the value proposed by p_i in the scalar version. The value it proposes to the vector version is simply the vector $[v_i, \ldots, v_i]$.

From the scalar version to the vector version The algorithm described in Figure 2 implements the vector version from the scalar version. A process p_i first proposes the vector $input_i$ (that contains its vector proposal) to each consensus instance of the underlying scalar version of the k-simultaneous consensus problem. It then obtains a pair $\langle c_i, w_i\rangle$ and decides on the pair $\langle c_i, d_i\rangle$ where d_i is the value in $w_i[c_i]$ (i.e., a value proposed by a process to the c_i-th consensus instance). The proof is easy and left to the reader. It is also easy to see that the size n of the set $\mathcal{I}$ of input values is the same in the vector version and the underlying scalar version.

operation $KSC.\mathsf{sc\_propose}_k(v_i^1, \ldots, v_i^k)$: % vector version %
(01) $input_i \leftarrow [v_i^1, \cdots, v_i^k]$;
(02) $\langle c_i, w_i\rangle \leftarrow KSC.\mathsf{sc\_propose}_k(input_i)$; % scalar ver. %
(03) **let** $d_i = w_i[c_i]$;
(04) $\mathsf{return}\langle c_i, d_i\rangle$.

Fig. 2 k-Simultaneous consensus: from the scalar version to the vector version

3 k-Set agreement vs k-simultaneous consensus

This section shows that the k-set agreement problem and the scalar k-simultaneous consensus problem are equivalent. To that end it presents two wait-free constructions, one in each direction. Both constructions are independent of the number of processes.

3.1 From scalar k-simultaneous consensus to k-set agreement

A pretty simple wait-free algorithm that builds a k-set agreement object (denoted KSA) on top of a k-simultaneous consensus object (denoted KSC) is described in Figure 3. The invoking process p_i calls the underlying object KSC with its input to the k-set agreement as input, and obtains a pair $\langle c_i, d_i\rangle$. It then returns d_i as the decision value for its invocation of $KSA.\mathsf{set\_propose}_k(v_i)$.

operation $KSA.\mathsf{set\_propose}_k(v_i)$:
(01) $\langle c_i, d_i\rangle \leftarrow KSC.\mathsf{sc\_propose}_k(v_i)$;
(02) $\mathsf{return}(d_i)$.

Fig. 3 From scalar k-simultaneous consensus to k-set agreement

Lemma 1 *The algorithm described in Figure 3 is a wait-free construction of a k-set agreement object from a scalar k-simultaneous consensus object.*

Proof The proof is immediate. The termination and validity of the k-set agreement object follow directly from the code and the same properties of the underlying k-simultaneous consensus object. The agreement property follows from the fact that at most k values can be decided from the k consensus instances of the k-simultaneous consensus object. $\square_{Lemma\ 1}$

3.2 From k-set agreement to scalar k-simultaneous consensus

A wait-free algorithm that constructs a scalar k-simultaneous consensus object KSC from a k-set agreement object KSA is described in Figure 4. ($|snap_i|$ denotes the number of elements in $snap_i$.)

In the algorithm, the processes first go through a k-set agreement object to reduce the number of distinct values to at most k (line 01). Then, each process p_i (1) posts the value it has just obtained in the cell $SM[i]$ of the shared memory (initialized to $\perp$), and (2) takes a snapshot of the whole shared memory (line 03). Finally, a process p_i returns the pair $\langle c_i, d_i\rangle$ where the consensus instance c_i is defined as the number of values in the set returned to p_i by its snapshot invocation, and d_i is the minimum value in that set.

```
operation KSC.sc_propose_k(v_i):
(01) dv_i ← KSA.set_propose_k(v_i);
(02) SM[i] ← dv_i;
(03) snap_i ← SM.set_snapshot();
(04) let c_i = |snap_i|; let d_i = minimum value in snap_i;
(05) return⟨c_i, d_i⟩.
```

Fig. 4 From k-set agreement to scalar k-simultaneous consensus

Lemma 2 *The algorithm described in Figure 4 is a wait-free construction of a scalar k-simultaneous consensus object from a k-set agreement object.*

Proof The code in Figure 4 is wait-free since there are no loops and both the k-set agreement and the snapshot operations are wait-free. The validity follows from the fact that all the values in the algorithm originate from process inputs.

Since the snapshots by the different processes define a linearizable sequence ordered by containment, they also define a non-decreasing sequence when we consider the size of the snapshots returned to the processes. Therefore, there is a unique snapshot value of a given size and hence the minimum value in each snapshot of a given size is unique. Thus there are at most k distinct snapshot sizes, each with its unique minimum value. Hence, there are at most k distinct outputs returned and any two processes that return a pair with the same snapshot size (same first coordinate) have the same value associated with it, which proves the agreement property of the k-simultaneous consensus. $\square_{Lemma\ 2}$

3.3 A first equivalence

Theorem 1 *The k-set agreement problem and the scalar k-simultaneous consensus problem (both with sets of possible input values of the same size n) are wait-free equivalent in read/write shared memory systems made up of an arbitrary number of processes.*

Proof The proof of the equivalence follows directly from Lemmas 1 and 2. $\square_{Theorem\ 1}$

4 Binary vs multivalued k-simultaneous consensus

The operation $\mathsf{bin\_sc\_propose}_k()$ is trivially a particular instance of the $\mathsf{sc\_propose}_k()$ operation: it corresponds to the case where only two values can be proposed ($\mathcal{I} = \{0, 1\}$). This section focuses on the transformation in the other direction. Assuming $|\mathcal{I}|$ is bounded and $n = |\mathcal{I}|$ is known to the processes, this section describes an algorithm that implements the scalar multivalued $\mathsf{sc\_propose}_k()$ operation from atomic registers and binary vector simultaneous consensus objects. Let us observe that, while every process knows n, no process knows initially the values that define the set $\mathcal{I}$ (it only knows the value it proposes).

4.1 A modular construction

An intermediary object The construction presented in the next subsection builds an intermediary object, that we call a *restricted ℓ-simultaneous consensus* object. The aim of such an object is to reduce by one the number of proposed values. More precisely, assuming that at most $\ell + 1$ different values are proposed by the processes, this object guarantees that (1) each process decides a value, and (2) at most ℓ different values are decided on. More formally, each of an arbitrary number of processes proposes a value such that at most $\ell + 1$ different values are proposed and the processes decide on at most ℓ different pairs $\langle c_i, d_i\rangle$, such that $1 \leq c_i \leq \ell$, each d_i is a value that has been proposed, and any two processes that return a pair with the same c_i also return the same d_i.

The next subsection (Section 4.2) shows how a restricted ℓ-simultaneous consensus object can be built out of atomic registers and a binary vector ℓ-simultaneous consensus object.

The construction Here we show how a cascading sequence of restricted ℓ-simultaneous consensus objects for $\ell = n-1, n-2, \ldots, k$ is used to construct a k-simultaneous consensus object KSC. Each restricted simultaneous consensus object in the sequence reduces the number of different values by one and the whole sequence reduces the size of the set of proposed values from n to k as described in Figure 5. Notice that a binary ℓ-simultaneous consensus is trivially implemented from binary k-simultaneous consensus for $\ell \geq k$, thus, all together we construct a multivalued k-simultaneous consensus from binary k-simultaneous consensus.

```
operation KSC.sc_propose_k(v_i):
(01) prop_i ← v_i;
(02) for ℓ from n − 1 step −1 to k do
(03)     ⟨c_i, prop_i⟩ ← RSC[ℓ].rsc_propose_ℓ(prop_i)
(04) end for;
(05) return⟨c_i, prop_i⟩.
```

Fig. 5 From restricted simultaneous consensus to scalar k-simultaneous consensus

Lemma 3 *The algorithm described in Figure 5 is a wait-free construction of a scalar k-simultaneous consensus object from restricted ℓ-simultaneous consensus objects, with $\ell = n-1, \cdots, k$.*

Proof The proof relies on the fact that the loop is made up of consecutive rounds. As there are initially at most n different values proposed by the processes, it follows from the definition of the $RSC[n-1]$ object that at most $n-1$ of these values are returned by the invocations $RSC[n-1].\mathsf{rsc\_propose}_{n-1}()$ issued by the processes. Then, the next rounds reduce the number of values to (at most) k. Finally, it follows from the definition of the last restricted simultaneous consensus object ($RSC[k]$) that the invocations $RSC[k].\mathsf{rsc\_propose}_k()$ return at most k pairs $\langle c_i, d_i\rangle$ and those are such that $1 \leq c_i \leq k$. As for any two pairs $\langle c_i, prop_i\rangle$ and $\langle c_j, prop_j\rangle$ we have $(c_i = c_j) \Rightarrow (prop_i = prop_j)$, the agreement property follows. The validity and (wait-free) termination properties follow directly from the text of the algorithm and the corresponding properties of the underlying $RSC[n-1..k]$ objects. $\square_{Lemma\ 3}$

4.2 Constructing a restricted ℓ-simultaneous consensus object

The construction The wait-free algorithm constructing a restricted ℓ-simultaneous consensus object is described in Figure 6. To reduce the number of values from $\ell+1$ to ℓ, the processes go through two sequential phases (lines 01-10, and lines 11-23). Only processes that have not decided in the first phase go into the second phase.

In the first phase (lines 01-10) the processes go through ℓ stages $T^1, ..., T^\ell$, each is one iteration of the loop in lines 02-09. A pair of arrays, $T1$ and $T2$, are associated with each stage r, $1 \leq r \leq \ell$; they are denoted $T1^r$ and $T2^r$. In each stage r, each process p_i posts its initial proposal (line 03) into $T1^r$, then takes a snapshot of the posted proposals (line 04), posts the set obtained from snapshot in the shared array $T2^r$ of snapshot values (line 05), and finally reads all the snapshot values deposited in $T2^r$ (line 06). If a process finds a snapshot of size 1 containing some value v_j but no snapshot of size 2 then it returns the pair $\langle c, v_j\rangle$, where c is the iteration number. Otherwise the process adopts the minimum value of some snapshot of size 2 or more and continues to the next iteration with this adopted value. p_i deterministically chooses a snapshot from which it adopts the minimum value, but which snapshot is chosen is unimportant, as long as the snapshot has at least two elements.

The key observation of the algorithm is that if a process has finished the ℓ iterations of the first phase without deciding (i.e., without returning in line 07 during any iteration), then there are snapshots of size 2 that have been posted in *all* the stages of the first phase. Let us notice that, due to the minimum function in line 08, one value is left behind in each iteration. Thus at most 2 different values arrive at the last (ℓ-th) iteration and, if some process did not decide in this last iteration, then this last size 2 snapshot is not empty. The size 2 snapshot in all the other iterations is also not empty because otherwise two values would have been left behind in one of the iterations, ensuring that all processes decide by the last iteration (See Lemma 5).

In the second phase (lines 11-23), all the processes that have not decided in the first phase use the vector version binary ℓ-simultaneous consensus object to decide on one of the values in these non-empty size 2 snapshots in a way that is consistent with all the decisions that have been already made during the first phase. For each stage of the first phase we associate the smaller value of the size 2 snapshot with 0, and the larger with 1. If the process also sees a snapshot of size 1 in stage r, then the r-th entry in its proposed vector is the binary value associated with the value in the size 1 snapshot (lines 14 and 15). Otherwise the process proposes an arbitrary binary value (say 0) for the r-th entry of its proposed binary vector (line 16)[3]. This ensures that a value that has been decided by some process during the stage r of the first phase will be the value proposed by all the processes that enter the second phase.

Finally, the binary ℓ-simultaneous consensus object is used (line 19) to decide on one of the values in these size 2 snapshots ($T2^r[2]$) and the algorithm terminates.

Proof The rest of this section formalizes the previous intuitive presentation by proving that the algorithm described in Figure 6 implements a restricted ℓ-simultaneous consensus object.

Each cell of the shared array $T1^r$ is written at most once. It is then read through $\mathsf{set\_snapshot}()$ operations, and the returned snapshots are posted in $T2^r$. Thus, the sets of values associated with each snapshot form a growing sequence and each set contains all previous sets in the sequence. Hence,

[3] Let us observe that the algorithm can easily be made fully deterministic. We write "some" at line 08 and "arbitrarily" at line 16 to emphasize the fact that the choice of the snapshot value (line 08) and the choice of a proposed binary value (line 16) are irrelevant for the correctness of the reduction. The replacement of "some" and "arbitrary" by deterministic statements does not modify the proof.

```
operation KSC.rsc_propose_ℓ(v_i):
(01)  est_i ← v_i;
(02)  for r from 1 to ℓ do
(03)      T1^r[i] ← est_i;
(04)      s_i ← set_snapshot(T1^r);
(05)      T2^r[|s_i|] ← s_i;
(06)      for j from 1 to ℓ + 1 do ss[j] ← T2^r[j] end for;
(07)      if (ss[1] = {v} ≠ ⊥) ∧ (ss[2] = ⊥)
          then return⟨r, v⟩
(08)      else est_i ← min(ss[x])
                  for some x such that (ss[x] ≠ ⊥ ∧ x ≥ 2)
(09)      end if ;
(10)  end for;
(11)  for each r ∈ {1, ··· , ℓ} do
(12)    let v_m = min(T2^r[2]); % smaller value in T2^r[2] %
(13)    let v_M = max(T2^r[2]; % larger value in T2^r[2]) %
(14)    case (T2^r[1] = {v_m})
                  then  prop_i[r] ← 0
(15)          (T2^r[1] = {v_M})
                  then  prop_i[r] ← 1
(16)              else  prop_i[r] ← 0 or 1 arbitrarily
(17)    end case
(18)  end for;
(19)  (c_i, dec_i) ← BSC[ℓ].bin_sc_propose_ℓ(prop_i);
                        % vector version %
(20)  if (dec_i = 1) then d_i ← max(T2^{c_i}[2])
(21)                 else d_i ← min(T2^{c_i}[2])
(22)  end if;
(23)  return⟨c_i, d_i⟩.
```

Fig. 6 From binary ℓ-simultaneous consensus to restricted ℓ-simultaneous consensus

Lemma 4 *For every $r, 1 \leq r \leq \ell$, for every $x \geq 1$, at most one set of values of size x is written in $T2^r[x]$ by the processes.*

The following lemma establishes that if a process does not decide in the first phase, a snapshot of size 2 has been posted in each stage r, $1 \leq r \leq \ell$ when the process starts the second phase.

Lemma 5 *In the second phase (Lines 11-23), for every $r, 1 \leq r \leq \ell$, each read of $T2^r[2]$ returns a non-$\perp$ value.*

Proof Let p_i be a process that does not decide in the first phase and starts executing the second phase. Let us assume for contradiction that the lemma is false. This means that, while p_i is executing the second phase of the protocol, a read of $T2^R[2]$ for some $R, 1 \leq R \leq \ell$ returns $\perp$. We show that p_i would have to decide in the first phase at line 07: a contradiction.

Write, read and set_snapshot() operations are linearizable. Let τ be the linearization point of the read of $T2^R[2]$ issued by p_i that returns $\perp$. Since no process writes $\perp$ in $T2^R[2]$, every read of $T2^R[2]$ linearized before τ must return $\perp$.

For every $r, 1 \leq r \leq \ell + 1$, let $I[r]$ be the set of values proposed before τ to the r-th iteration in the first phase of the protocol. We say that a value v is proposed to iteration r before τ if v is written in some entry of $T1^r$ and the corresponding write operation is linearized before τ. We claim that for every $r, 1 \leq r < \ell$, $|I[r] - I[r+1]| \geq 1$, i.e., each iteration eliminates at least one initial value (Claim *C*). Since at most $\ell + 1$ values are initially proposed, Claim *C* implies that at most $\ell + 1 - (R-1) = \ell - R + 2$ values can be written in $T1^R$. Assuming Claim *C* (which is proved in the sequel) we consider below the prefix of the execution that ends at time τ. The proof is divided in two cases according to the value of R.

- $R = \ell$. Following Claim *C* at most two values are written in $T1^\ell$, no snapshot of size ≥ 3 can be posted in $T2^\ell$. Process p_i executes iteration R before time τ. In particular, its read of $T2^\ell[2]$ returns $\perp$. It then follows from the code that the snapshot of $T1^\ell$ by p_i contains a single value, from which we conclude that p_i decides at Line 07 since it observes no posted snapshot of size ≥ 2 in $T2^\ell$.
- $R < \ell$. Each value written in $T1^{R+1}$ is the smallest value in some snapshot of size ≥ 2 that have been posted in $T2^R$. We know that at most $(\ell + 1) - (R - 1)$ values are written in $T1^R$. Therefore, no snapshot of size $> (\ell + 1) - (R - 1)$ is posted in $T2^R$. Moreover, before τ, no snapshot of size 2 is observed.
  Furthermore, it follows from Lemma 4 that for every $x, 3 \leq x \leq (\ell + 1) - (R - 1)$, at most one snapshot of size x can be observed in $T2^R$. Finally, since each snapshot defines a unique estimate, we conclude that at most $(\ell + 1) - (R - 1) - 2 = \ell - R$ values are proposed to iteration $R + 1$, i.e., $|I[R+1]| \leq \ell - R$.
  It remains to show that p_i decides in the first phase. By applying Claim C to iterations $R+1, \ldots, \ell-1$, we have $|I[\ell]| \leq 1$. Process p_i executes iteration ℓ before τ. Therefore, p_i obtains a snapshot of size 1, writes it in $T2^\ell[1]$ and then decides since no snapshot of size 2 is posted in $T2^\ell$ before τ.

Claim C: $\forall r, 1 \leq r \leq \ell - 1, |I[r] - I[r+1]| \geq 1$.
Proof of Claim C. A value written in $T1^{r+1}$ is the smallest value in some snapshot of size ≥ 2 posted in $T2^r$ (Line 08). The claim follows since there are at most $|I[r]| - 1$ distinct snapshots of size ≥ 2 that may be written in $T2^r$. *End of the Proof of Claim C.* $\square_{Lemma\ 5}$

Lemma 6 *If a process decides $\langle r, v \rangle$, then $r \in \{1, \ldots, \ell\}$ and v is a value proposed by a process.*

Proof The fact that $r \in \{1, \ldots, \ell\}$ follows directly from the code of the algorithm. The validity of v fol-

lows from the observation that a value enters a snapshot only if it was already in a previous snapshot, or was proposed by a process during the first stage of the first phase. $\square_{Lemma\ 6}$

Lemma 7 *If p_i and p_j decide $\langle r_i, v_i\rangle$ and $\langle r_j, v_j\rangle$, respectively, we have $(r_i = r_j) \Rightarrow (v_i = v_j)$.*

Proof For each consensus instance R, let D_R denote the set of processes that decide in the R-th consensus instance. We consider three cases according to the phase(s) in which processes that belong to D_R decide.

- All the processes that belong to D_R decide in the first phase (Line 07). In that case, process $p_i \in D_R$ decides a value contained in a singleton snapshot that it has observed in $T2^R$. Agreement follows from the fact that a unique snapshot of size one may be posted in $T2^R$ by the different processes (Lemma 4).
- All the processes that belong to D_R decide in the second phase (line 23). Each process $p_i \in D_R$ gets back a pair $\langle R, d_i\rangle$ from the binary ℓ-simultaneous consensus object. Due to the agreement property of the object, $\exists d \in \{0, 1\}$ such that $\forall p_i \in D_R, d_i = d$.
  Moreover, per Lemma 5, every process in D_R observes a snapshot of size 2 in $T2^r$ at lines 20 or 21 and, by Lemma 4, they observe the same snapshot. It then follows from lines20-22 that each process in D_R returns the same value.
- Decisions occur in both phases. Let C be the set of processes that invoke the binary ℓ-simultaneous consensus object (a process that belongs to C could have not decided in the first phase). Among them, let p_c be the first process that reads $T2^R[1]$ in the second phase of the algorithm (lines 14-15). This occurs at time τ. There are two cases according to the value returned by that read.
  - Suppose that p_c does not observe a snapshot of size 1 ($T2^R[1] = \perp$). In that case no process in D_R could decide in the first phase of the algorithm.
    Assume for contradiction that process p_i decides $\langle R, v\rangle$ at line 07. p_i must observe $T2^R[1] = \{v\}$ at some time τ'. As the read of $T2^R[1]$ by p_c returns $\perp$, and no process writes $\perp$ in $T2^R[1]$, it follows that $\tau' > \tau$. But by Lemma 5, we know that $T2^R[2] \neq \perp$ when p_c starts the second part of the protocol. Consequently, p_i must also observe $T2^R[2] \neq \perp$, which prevents it from deciding in the first part of the protocol (line 07).
  - Suppose that p_c observes a singleton snapshot $\{v\}$ in $T2^R$. Per Lemma 4, only one singleton snapshot can be written in $T2^R$. Therefore, every process in C reads $\{v\}$ in $T2^R[1]$ at lines 14-15. Then every process in C "proposes" v to the Rth binary-consensus. More precisely, each process proposes $d = 0$ (resp. $d = 1$) to the R-th binary consensus if v is the smallest (resp. greatest) value in the snapshot written in $T2^R[2]$ (by Lemma 5, there is always a snapshot s of size 2 written in $T2^R[2]$ when processes execute the second part of the protocol. Since snapshots are ordered by containment, $v \in s$).
    Therefore, by the validity property of the ℓ-simultaneous binary consensus object, each process in D_R that decides in the second part gets back $\langle R, d\rangle$ from the object, and consequently returns the same pair $\langle R, v\rangle$ (lines 20-22). Moreover, a process in D_R that decides in the first part of the protocol returns also $\langle R, v\rangle$, $\{v\}$ being the only snapshot written in $T2^R[1]$.

$\square_{Lemma\ 7}$

Lemma 8 *The algorithm described in Figure 6 is a wait-free construction of a restricted ℓ-simultaneous consensus object from a binary vector ℓ-simultaneous consensus object for any number of processes.*

Proof The wait-free property follows directly from the text of the algorithm and the same property of the underlying binary simultaneous consensus object. The validity and the agreement properties have been proved in Lemma 6 and Lemma 7, respectively. $\square_{Lemma\ 8}$

4.3 A second equivalence

Theorem 2 *The multivalued k-simultaneous consensus problem (where the size n of the set of the possible input values is known by the processes), and the binary k-simultaneous consensus problem are wait-free equivalent in read/write shared memory systems made up of an arbitrary number of processes.*

Proof As already indicated, the multivalued version of the problem trivially solves its binary version. The other direction follows from the algorithm described in Figure 5 (proved in Lemma 3), and the algorithm described in Figure 6 (proved in Lemma 8). $\square_{Theorem\ 2}$

5 Conclusion

This paper has introduced and studied the k-simultaneous consensus problem. Its main result is the following the-

orem, whose proof follows from Theorem 1 and Theorem 2.

Theorem 3 *The k-set agreement problem and the k-simultaneous binary consensus problem are wait-free equivalent in asynchronous read/write shared memory systems made up of an arbitrary number of processes.*

This theorem provides a new characterization of the k-set agreement problem. This characterization shows that k-simultaneous consensus captures both k-set agreement and consensus.

The paper has focused mainly on establishing equivalence between several variants of the simultaneous consensus problem and the set-agreement problem. It leaves open several avenues for future research, some of which are detailed below.

Generalization to other decision problems Given a decision problem (task) T, the k-simultaneous version of T can be defined in a way similar to the k-simultaneous consensus problem. Instead of k instances of the consensus problem, we then consider k instances of T. Each process proposes a value to each instance of T and is required to decide in at least one of the k instances. Of course, a value decided in an instance must comply with the specification of T.

As a simple example, let us consider the following natural generalization of the k-simultaneous consensus problem that is the "k-simultaneous ℓ-set-agreement" [3, 13]. This problem is defined in the same way as the k-simultaneous consensus problem, namely, each process has to decide a pair $\langle c, v\rangle$ subject to the following constraints: (1) $1 \leq c \leq k$, (2) v is a proposed value for the c-th instance and (3) at most ℓ values are decided in each instance. It is easy to see that the scalar version and the vector version of this problem are equivalent. Also, given a solution to the k-simultaneous ℓ-set-agreement problem, it easy to solve (ℓk)-set-agreement, since at most ℓk pairs are decided.

What about the other direction ? A simple modification of the algorithm described in Figure 4 constructs a k-simultaneous ℓ-set-agreement object from an (ℓk)-set-agreement object. The first statement in line 04 that defines the consensus instance is replaced by "**let** $c_i = \lceil \frac{|snap_i|}{\ell} \rceil$" that now defines the k-set instance number associated with the value decided by p_i.

The (ℓk)-set-agreement object reduces the number of distinct values to ℓk. Thus, the first coordinate of the decided pairs is at most k. Finally, as the sets of values obtained by snapshot invocations are related by containment, there are at most ℓ distinct sets $snap$ such that $(c-1)\ell + 1 \leq |snap| \leq c\ell$. Therefore, at most $k\ell$ values are decided in the c-th instance. Hence,

Theorem 4 *The (ℓk)-set agreement problem and the k-simultaneous ℓ-set-agreement problem are wait-free equivalent in asynchronous read/write shared memory systems made up of an arbitrary number of processes.*

Cost of the equivalences The algorithms presented in Section 3 use only one extra object in addition to atomic registers. For example, the algorithm described in Figure 4 shows that only one k-set-agreement object is needed to solve the multivalued scalar k-simultaneous consensus problem.

The second construction from binary k-simultaneous consensus to multivalued k-simultaneous consensus uses $(n-k)$ binary simultaneous consensus objects, where n is the size of the set $\mathcal{I}$ of the possible input values. By contrast, as far as we know, the best construction of a multivalued consensus object from binary consensus objects requires only $\log(n)$ base binary consensus objects (e.g., [21]).

Another interesting open problem concerns the improvement of the step complexity of the algorithm presented in Figure 6 that builds a restricted ℓ-simultaneous consensus object from binary ℓ-simultaneous consensus objects.

The case of message-passing systems The paper focused on the shared memory model. Another interesting open problem is: are Theorems 1, 2 and 3 still valid in an asynchronous message-passing system prone to process crashes? If the number N of processes is fixed and known, the answer is yes if at most $t < N/2$ processes may crash (this is because atomic registers can be implemented in such systems [2]). For larger values of t, "Are the k-set-agreement problem and the binary k-simultaneous problem equivalent?" remains an open question.

Acknowledgements We would like to thank the anonymous referees for their careful reading and their suggestions that help improve both the content and the presentation of the paper.

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. Attiya H., Bar-Noy A., and Dolev D., Sharing Memory Robustly in Message-Passing Systems. *Journal of the ACM*, 42(1):124-142, 1995.
3. Afek Y., Gafni E., Rajsbaum S., Raynal M. and Travers C., Simultaneous Consensus Tasks: a Tighter Characterization of Set Consensus. *Proc. 8th Int'l Conference on Distributed Computing and Networking (ICDCN'06)*, Springer-Verlag LNCS #4308, pp. 331-341, 2006.

4. Aguilera M., A pleasant stroll through the land of infinitely many creatures. *ACM SIGACT News, Distributed Computing Column,* 35(2):36-59, 2004.
5. Attiya H. and Welch J., *Distributed Computing: Fundamentals, Simulations and Advanced Topics,* (2d Edition), Wiley-Interscience, 414 pages, 2004.
6. Borowsky E. and Gafni E., Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. *Proc. 25th ACM Symposium on Theory of Computing (STOC'93)*, pp. 91-100, 1993.
7. Chaudhuri S., More *Choices* Allow More *Faults:* Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation,* 105:132-158, 1993.
8. Dolev D., Reischuk R. and Strong R., Early Stopping in Byzantine Agreement. *Journal of the ACM*, 37(4):720-741, April 1990.
9. Dwork C. and Moses Y., Knowledge and Common Knowledge in a Byzantine Environment: Crash Failures. *Information and Computation*, 88(2):156-186, 1990.
10. 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.
11. Gafni E., Merritt M., and Taubenfeld G., The Concurrency Hierarchy, and Algorithms for Unbounded Concurrency. *Proc. 20th ACM Symposium on Principles of Distributed Computing (PODC'01)*, ACM Press, pp. 161-170, 2001.
12. Gafni E. and Rajsbaum S., Musical Benches. *Proc. 19th Int'l Symposium on Distributed Computing (DISC'05)*, Springer Verlag LNCS #3724, pp. 63–77, 2005.
13. Gafni E., Rajsbaum R., Raynal M. and Travers C., The Committee Decision Problem. *Proc. 8th Latin American Theoretical Informatics (LATIN'06)*, Springer-Verlag LNCS #3887, pp. 502-514, 2006.
14. Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991.
15. Herlihy M.P. and Shavit N., The Topological Structure of Asynchronous Computability. *Journal of the ACM*, 46(6):858-923, 1999.
16. 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.
17. Lamport L., On interprocess communication, Part 1: Models, Part 2: Algorithms. *Distributed Computing*, 1(2):77-101, 1986.
18. Loui M.C., Abu-Amara H., Memory Requirements for Agreement Among Unreliable Asynchronous Processes. *Advances in Computing research*, JAI Press, 4:163-183, 1987.
19. Lynch N.A., Distributed Algorithms. *Morgan Kaufmann Pub.*, San Francisco (CA), 872 pages, 1996.
20. Merritt M. and Taubenfeld G., Computing with Infinitely Many Processes. *Proc. 14th Int'l Symposium on Distributed Computing (DISC'00)*, Springer-Verlag LNCS #1914, pp. 164-178, 2000.
21. Mostéfaoui A., Raynal M. and Tronel F., From Binary Consensus to Multivalued Consensus in Asynchronous Message-Passing Systems. *Information Processing Letters*, 73:207-213, 2000.
22. Saks M. and Zaharoglou F., Wait-Free k-Set Agreement is Impossible: The Topology of Public Knowledge. *SIAM Journal on Computing*, 29(5):1449-1483, 2000.
23. Zielińsky P., Anti-Ω: the Weakest Failure Detector for Set Agreement. *Proc. 27th ACM Symposium on Principles of Distributed Computing (PODC'08)*, ACM Press, pp. 55-64, 2008.