Set-Consensus Collections are Decidable

Carole Delporte-Gallet[1], Hugues Fauconnier[2], Eli Gafni[3], and Petr Kuznetsov[*4]

1 **IRIF, Université Paris-Diderot, Paris, France**
cd@liafa.jussieu.fr
2 **IRIF, Université Paris-Diderot, Paris, France**
hf@liafa.jussieu.fr
3 **UCLA, Los Angeles, CA, USA**
eli@ucla.edu
4 **Télécom ParisTech, Paris, France**
petr.kuznetsov@telecom-paristech.fr

Abstract

A natural way to measure the power of a distributed-computing model is to characterize the set of tasks that can be solved in it. In general, however, the question of whether a given task can be solved in a given model is undecidable, even if we only consider the wait-free shared-memory model. In this paper, we address this question for restricted classes of models and tasks. We show that the question of whether a collection C of (ℓ, j)-*set consensus* objects, for various ℓ (the number of processes that can invoke the object) and j (the number of distinct outputs the object returns), can be used by n processes to solve wait-free k-set consensus is decidable. Moreover, we provide a simple $O(n^2)$ decision algorithm, based on a dynamic programming solution to the Knapsack optimization problem. We then present an *adaptive* wait-free set-consensus algorithm that, for each set of participating processes, achieves the best level of agreement that is possible to achieve using C. Overall, this gives us a complete characterization of a read-write model defined by a collection of set-consensus objects through its *set-consensus power*.

1998 ACM Subject Classification E.1.2 Distributed Data Structures, C.2.4 Distributed Systems

Keywords and phrases Decidability, distributed tasks, set consensus, Knapsack problem

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.7

1 Introduction

A plethora of models of computation were proposed for distributed environments. The models vary in timing assumptions they make, types of failures they assume, and communication primitives they employ. It is hard to say *a priori* whether one model provides more power to the programmer than the other. A natural way to measure this power is to characterize the set of distributed tasks that can be solved in a model. In general, however, the question of whether a given task can be solved in the popular *wait-free* read-write model, i.e., tolerating asynchrony and failures of arbitrary subsets of processes, is undecidable [13]. Of course, in models in which processes can additionally access arbitrary objects, the question is not decidable either. However, many natural models have been shown to be characterized by their power to solve set consensus [10].

* The author was supported by the Agence Nationale de la Recherche, under grant agreement N ANR-14-CE35-0010-01, project DISCMAT.

20th International Conference on Principles of Distributed Systems (OPODIS 2016).
Editors: Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone; Article No. 7; pp. 7:1–7:15
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper, we consider models in which n completely asynchronous processes communicate through reads and writes in the shared memory and, in addition, can access *set-consensus* objects. An (ℓ, j)-set-consensus object solves j-set consensus among ℓ processes, i.e., the object can be accessed by up to ℓ processes with *propose* operations that take natural numbers as inputs and return natural numbers as outputs, so that the set of outputs is a subset of inputs of size at most j. Set consensus is a generalization of consensus and, like consensus [18], exhibits a *universailty* property: ℓ processes can use (ℓ, j)-set consensus and read-write registers to implement j state machines, ensuring that at least one of them makes progress [12]. In this paper, we explore what level of agreement, and thus "degree of universality", can be achieved using any number of objects from a given set-consensus collection.

The special case when only one type of set consensus can be used in the implementation was resolved in [4, 8, 23]. Assuming that $k \geq j\lceil n/\ell \rceil$, we trivially solve $j\lceil n/\ell \rceil$-set consensus, by splitting n processes into $\lceil n/\ell \rceil$ groups of size ℓ (or less). A slightly more complex converse bound [4, 8, 23], accounting for the "delta" between n and $\ell\lceil n/\ell \rceil$, resolves the special case when only one type of set consensus object can be used.

Characterizing a general model in which processes communicate via objects in an arbitrary collection C of possibly different set-consensus objects is more difficult. For example, let C be $\{(2, 1), (5, 2)\}$, i.e., every 2 processes in our system can solve consensus and every 5 can solve 2-set consensus. What is the best level of agreement we can achieve using registers and an arbitrary number of objects in C in a system of 9 processes? One can easily see that 4-set consensus can be solved: the first two pairs of processes solve consensus and the remaining 5 invoke 2-set consensus, which would give at most 4 different outputs. One can also let the groups of the first 5 and the remaining 4 each solve 2-set consensus. (In general, any two set-consensus objects (ℓ_1, j_1) and (ℓ_2, j_2) can be used to solve $(\ell_1 + \ell_2, j_1 + j_2)$-set consensus.) But could we do $(9, 3)$-set consensus with C?

We propose a simple way to characterize the power of a set-consensus collection. By convention, let (ℓ_0, j_0) be $(1, 1)$, and note that $(1, 1)$-set consensus is trivially solvable. We show that a collection $C = \{(\ell_0, j_0), (\ell_1, j_1), \ldots, (\ell_m, j_m)\}$ solves (n, k)-set consensus if and only if there exist $x_0, x_1, \ldots, x_m \in \mathbb{N}$, such that $\sum_i \ell_i x_i \geq n$ and $\sum_i j_i x_i \leq k$. Thus, determining the power of C is equivalent to solving a variation of the Knapsack optimization problem [21], where each j_i serves as the "weight" of an element in C, i.e., how much disagreement it may incur, and each ℓ_i serves as its "value", i.e., how many processes it is able to synchronize. We describe a simple $O(n^2)$ algorithm for computing the power of C for solving set consensus among n processes using the dynamic programming approach.

The sufficiency of the condition is immediate. Indeed, the condition implies that we can partition the set of n processes in $\sum_i x_i$ groups: x_0 groups of size (at most) ℓ_0, x_1 groups of size (at most) ℓ_1, …, x_m groups of size (at most) ℓ_m. Each of the x_i groups of size ℓ_i, $i = 0, \ldots, m$, can independently solve j_i-set consensus using a distinct (ℓ_i, j_i)-set-consensus object in C, which gives us at most $\sum_i j_i x_i \leq k$ different outputs in total.

The necessity uses a generalized version of the BG simulation [3, 5] that allows to simulate, in the read-write shared-memory model, a protocol that uses various types of set-consensus objects. We use this simulation to show that if a collection not satisfying the condition solves (n, k)-set consensus, then $k + 1$ processes can solve k-set consensus using read-write registers, contradicting the classical wait-free set-consensus impossibility result [3, 20, 24]. Interestingly, the necessity of this condition holds even if we can use read-write registers in addition to the elements in C.

Thus, we derive a complete characterization of models defined by collections of set-consensus objects. In particular, it allows us to determine the *j-set-consensus number* of a set-consensus collection C as the maximal number of processes that can achieve j-set consensus using C and read-write registers. Applied to arbitrary objects, this metric is a natural generalization of Herlihy's consensus number [18].

Coming back to the collection $C = \{(2,1),(5,2)\}$, our characterization implies that 4 is the best level of set consensus that can be achieved by 9 processes with C. Observe, however, that if only 2 processes participate, then they can use C to solve consensus, i.e., to achieve "perfect" agreement. Applying our condition, we also see that participating sets of sizes 3 up to 5 can solve 2-set consensus, participating sets of size 6 up to 7 can solve 3-set consensus, participating sets of size 8 up to 10 can solve 4-set consensus, etc. That is, for every given participating set, we can devise an *optimal* set-consensus algorithm that ensures the best level of agreement achievable with C.

An immediate question is whether we could *adapt* to the participation level and ensure the best possible level of agreement in any case? Such algorithms are very useful in large-scale systems with bounded contention levels. We show that this is possible by presenting an *optimally adaptive* set-consensus algorithm. Intuitively, for the currently observed participation, our algorithm employs the best algorithm and, in case the participating set grows, seamlessly relaxes the agreement guarantees by switching to a possibly less precise algorithm when there is a larger set of participants.

Our results thus imply that there is an efficient algorithm to decide whether one model defined by a collection of set-consensus object types can be implemented in model defined by another collection of set-consensus objects. We conjecture that the ability of any "reasonable" (yet to be defined precisely) shared-memory system to solve set consensus, captured by its j-set-consensus numbers, for all positive j, characterizes precisely its computing power with respect to solving tasks or implementing deterministic objects.

This work contributes to the idea that there is nothing special about consensus that set consensus cannot do. Indeed, set-consensus collections are decidable in the same way collections of consensus objects are [18]: the power of a collection of consensus objects $\{(\ell_1,1),(\ell_2,1),\ldots,(\ell_m,1)\}$ to solve consensus is determined by $\max_i \ell_i$. Furthermore, it was recently shown that the computational power of a class of deterministic objects cannot be characterized by its ability to solve consensus [2], which suggests the use of set consensus in a characterization. We see this paper as the first step towards proving the conjecture that the computational power of a deterministic object can be captured by its *set-consensus* number, determining the best level of agreement the object can reach for each given system size.

Roadmap. The rest of the paper is organized as follows. In Section 2, we recall the basic model definitions and simulation tools. In Section 3, we present and prove our characterization of set-consensus collections, and describe an efficient algorithm to compute the characterizing criterion. In Section 4, we present an adaptive algorithm that achieves the optimal level of agreement for each set of active participants having access to a given set-consensus collection. We discuss related work in Section 5 and conclude in Section 6.

2 Preliminaries

In this section, we briefly state our system model, recall the notion of a distributed task, and sketch the basic simulation tools that we use in the paper.

Processes and tasks. We consider a system Π of asynchronous processes that communicate via shared memory abstractions. We assume that process may only fail by crashing, and otherwise it must respect the algorithm it is given. A *correct* process never crashes. Shared abstractions we consider here include an *atomic-snapshot* memory [1] and a collection of objects solving *distributed tasks* [20].

An atomic-snapshot memory stores a vector of $|\Pi|$ values, one value per process, and exports atomic operations *update* and *snapshot*: operation *update*(p, v) performed by process p writes v in position p in the vector, and operation *snapshot*() returns the vector. Atomic-snapshot memory can be implemented, in a wait-free and linearizable manner, in the standard read-write shared-memory model [1].

A process invokes a task with an input value and the task returns an output value, so that the inputs and the outputs across the processes invoked the task respect the task specification and every correct process that participates decides (gets an output). More precisely, a *task* is defined through a set $\mathcal{I}$ of input vectors (one input value for each process), a set $\mathcal{O}$ of output vectors (one output value for each process), and a total relation $\Delta : \mathcal{I} \mapsto 2^{\mathcal{O}}$ that associates each input vector with a set of possible output vectors. An input $\perp$ denotes a *non-participating* process and an output value $\perp$ denotes an *undecided* process.

For vectors S and S' in $\mathcal{I}$ (resp., $\mathcal{O}$), we write $S \geq S'$ if S' is obtained from S by replacing some entries with $\perp$. We assume that if $\mathcal{I}$ (resp., $\mathcal{O}$) contains a vector S, then $\mathcal{I}$ (resp., $\mathcal{O}$) also contains any vector S' such that $S \geq S'$. We stipulate that if $(I, O) \in \Delta$, then (1) for all i, if $I[i] = \perp$, then $O[i] = \perp$, (2) for each O', such that $O \geq O'$, $(I, O') \in \Delta$ and, (3) for each I' such that $I' \geq I$, there exists some O' such that $O' \geq O$ for all i, if $I'[i] \neq \perp$, then $O'[i] \neq \perp$, and (I', O') in Δ.

An algorithm *solves a task* $T = (\mathcal{I}, \mathcal{O}, \Delta)$ *in a wait-free manner* if it ensures that in every execution in which processes start with an input vector $I \in \mathcal{I}$, every correct process decides, and the set of decided values, taken together with the processes taking these decisions, form a vector $O \in \mathcal{O}$ (where positions of non-decided processes are assigned $\perp$) such that $(I, O) \in \Delta$.

The task of k-set consensus. In the task of *k-set consensus*, input values are in a set of values V ($|V| \geq k + 1$), output values are also in V, and for each input vector I and output vector O, $(I, O) \in \Delta$ if the set of non-$\perp$ values in O is a subset of values in I of size at most k. The special case of 1-set consensus is called *consensus* [11]. More generally, *(ℓ, k)-set-consensus objects* ($k \leq \ell$) allow arbitrary subset of ℓ processes to solve k-set consensus.

Note that k-set consensus is an example of a *colorless* task (also known as a *convergence* task [5]): processes are free to use each others' input and output values, so the task can be defined in terms of input and output *sets* instead of vectors. Formally, let *val*(U) denote the set of non-$\perp$ values in a vector U. In a colorless task, for all input vectors I and I' and all output vectors O and O', such that $(I, O) \in \Delta$, *val*$(I) \subseteq$ *val*(I') and *val*$(O') \subseteq$ *val*(O), we have $(I', O') \in \Delta$. To solve a colorless task, it is sufficient to find an algorithm that allows just one process to decide. Indeed, if such an algorithm exists, we can simply convert it into an algorithm that allows every correct process to decide: every process simply applies the decision function to the observed state of any process that has decided and adopts the decision.

In contrast, (ℓ, k)-set consensus is not colorless in a system of $n > \ell$ processes, as it does not always allow a process to adopt the decision of another process: e.g., if a process does not belong to a set S of ℓ processes, it cannot provide outputs for j-set consensus for S.

Simulation tools. An execution of a given algorithm $\mathcal{A}$ by the processes $p_1, \dots p_n$ can be *simulated* by a set of *simulator* processes $s_1, \dots, s_\ell$ (or, simply, simulators) that run a distributed algorithm "mimicking" the steps of $\mathcal{A}$ in a *consistent* way. Informally, for every execution E_s of the simulation algorithm, there exists an execution E of $\mathcal{A}$ by $p_1, \dots, p_n$ such that the sequence of states simulated for every process p_i in E_s is observed by p_i in E.

A basic building block of our simulations is an *agreement* protocol [3, 5] that can be seen as a safe part of consensus. It exports one operation *propose*() taking $v \in V$ as a parameter and returning $w \in V$, where V is a (possibly infinite) *value set*. When a process p_i invokes *propose*(v) we say that p_i *proposes* v, and when the invocation returns v' we say that p_i *decides on* v'. Agreement ensures four properties:

(i) every decided value has been previously proposed,
(ii) no two processes decide on different values, and
(iii) if every participating process takes enough steps then eventually every correct participating process decides.

Here a process is called participating if it took at least one step in the computation. In fact, the agreement protocol in [3, 5] ensures that if every participating process takes at least three shared memory steps then eventually every correct participating process decides. If a participating process fails in the middle of an agreement protocol, then no process is guaranteed to return.

A generalized version of the agreement protocol, ℓ-*agreement* [4, 8], relaxes safety properties of agreement but improves liveness. Formally, in addition to (i) above, ℓ-agreement ensures:

(ii′) at most ℓ different values can be decided, and
(iii′) every correct participating process is guaranteed to decide, unless ℓ or more participating processes do not take enough steps.

Clearly, the agreement protocol we defined above is 1-agreement. An ℓ-agreement protocol with a proof (only sketched in [4, 8]) can be found in Reiners' thesis [23]. For completeness, given that the thesis is not easy to find, we present the proof in Appendix A.

3 A characterization of set-consensus collections

In this section, we introduce the notion of *agreement level* for a given set-consensus collection C and a given system size. Then we show that the metrics captures the power of C for solving set consensus. Then we show how to efficiently compute the agreement level of a given collection.

3.1 Agreement levels of C

Consider a model in which processes can communicate via an atomic-snapshot memory and set-consensus objects from a collection C. For brevity, we represent C as a set $\{(\ell_0, j_0), (\ell_1, j_1), \dots, (\ell_m, j_m)\}$ such that for each $i = 0, \dots, m$, the task of (ℓ_i, j_i)-set consensus can be solved $(\ell_i \geq j_i)$.

By convention, we assume that $(\ell_0, j_0) = (1, 1)$ is always contained in a collection C: $(1, 1)$-set consensus is trivially solvable. Note that (ℓ, j)-set consensus also solves (ℓ', j')-set consensus for all $\ell' \leq \ell$ and $j' \geq j$. Thus, without loss of generality, we can assume that the sequence $(\ell_1, j_1), \dots, (\ell_m, j_m)$ is monotonically increasing: $\ell_0 < \ell_1$ and for all $i = 1, \dots, m-1$, $\ell_i < \ell_{i+1}$ and $j_i < j_{i+1}$. (since we required that $(\ell_0, j_0) = (1, 1)$, there can be two elements of the type $(-, 1)$). In particular, for all n, C contains at most n elements (ℓ, j) such that $\ell \leq n$.

▶ **Definition 1** (Agreement level). Let $C = \{(\ell_0, j_0), (\ell_1, j_1), \ldots, (\ell_m, j_m)\}$ be a collection of set-consensus objects. The *agreement level for n processes of C*, denoted AL_n^C, is defined as:

- $\min \sum_i j_i x_i$
- under the constraints: $\sum_i \ell_i x_i \geq n$, $x_0, \ldots, x_m \in \{0, \ldots, n\}$

One can also interpret AL_n^C as the lowest k for which there exists a *multiset* $S = \{(t_1, s_1),$ …, $(t_p, s_p)\}$ of elements in C such that $\sum_i s_i = k$ and $\sum_i t_i \geq n$.[1]

3.2 Agreement levels and set consensus

We now can define a simple criterion to determine whether the model defined by C can solve (n, k)-set consensus. The criterion is *sufficient*, i.e., every model equipped with C that satisfies the criterion solves (n, k)-set consensus, and *necessary*, i.e., every model equipped with C that solves (n, k)-set consensus satisfies the criterion.

▶ **Theorem 2.** *(n, k)-set consensus can be solved using read-write registers and any number of objects taken in a set-consensus collection C if and only if $AL_n^C \leq k$.*

Proof. Suppose that $AL_n^C \leq k$. Thus, there exists a multiset $S = \{(t_1, s_1), \ldots, (t_p, s_p)\}$ of elements in C such that $\sum_i s_i \leq k$ and $\sum_i t_i \geq n$. We show how n processes can solve k-set consensus using S. Every p_i, $i = 1, \ldots, n$, is assigned to the element $(t_j, s_j) \in S$ such that $\sum_{\ell=1,\ldots,j-1} t_\ell < i \leq \sum_{\ell=1,\ldots,j} t_\ell$, invokes the assigned object of (t_j, s_j)-set consensus with its input and returns the corresponding output. Since $\sum_i s_i \leq k$, the total number of outputs does not exceed k.

Now suppose that C can be used to solve (k, n)-set consensus and let A be the corresponding algorithm. By contradiction, suppose that no multiset S satisfying the conditions above exists for C. Thus, for any multiset $\{(t_1, s_1), \ldots, (t_p, s_p)\}$ of elements in C such that $\sum_i s_i \leq k$, we have $\sum_i t_i < n$.

We show that we can then use a simulation of A to solve $(k + 1, k)$-set consensus using only read-write memory, contradicting the classical impossibility result [3, 20, 24]. The simulation we describe below is an extension of the *BG simulation* [3, 5], inspired by the algorithms described in [4, 8].

Simulation. Let $q_1, \ldots, q_{k+1}$ be a set of $k + 1$ simulator processes communicating via an atomic-snapshot memory. In its position in the snapshot memory, every simulator q_i maintains its estimate of the current simulated state of every simulated process in $\{p_1, \ldots, p_n\}$.

Note that the state of each p_ℓ (in algorithm A) unambiguously determines the next step that p_ℓ is going to take in the simulation, which can be an update operation, a snapshot operation, or an access to a (t, s)-set-consensus object. Since each update operation by p_ℓ is implicitly simulated by registering the last simulated state of p_ℓ in the shared memory, the simulators only need to explicitly simulate snapshot operations and accesses to set-consensus objects.

We associate each state of p_ℓ (assuming distinct local states) with a distinct agreement protocol (cf. Section 2), depending on the next step p_ℓ is going to take in that state:

- For a snapshot operation, we use one instance of the agreement (1-agreement) algorithm.
- For an access to a (t, s)-set-consensus object, we use one instance of s-agreement and one instance of 1-agreement.

[1] Note that assuming that $(1, 1) \in C$ implies $AL_n^C \leq n$.

The initial state of each simulated process is associated with a 1-agreement protocol.

The simulation proceeds in asynchronous rounds. In each round, a simulator q_i picks up the next simulated process p_ℓ in a round-robin fashion. To simulate a step of p_ℓ, q_i takes a snapshot of the memory and computes p_ℓ's latest simulated state by choosing the latest simulated state of p_ℓ found in the snapshot.

If p_ℓ is in the initial state, q_i invokes the agreement protocol (1-agreement) to compute the input of p_ℓ in the simulated run, using its input value (for k-set consensus) as a proposed value. Otherwise, q_i invokes the corresponding agreement protocol:

- To simulate a snapshot operation, q_i invokes the corresponding 1-agreement protocol, proposing the just read simulated system state (the vector of the latest simulated states of processes $p_1, \ldots, p_n$) as the outcome of the simulated snapshot.

  Recall that a simulator that has started but not finished the 1-agreement protocol for a given snapshot operation may block the simulated process forever. However, since the faulty simulator may be involved in at most one agreement protocol at a time, it can block at most one simulated process.

- To simulate an access of a (t, s)-set-consensus object, the simulator invokes the corresponding s-agreement protocol proposing p_ℓ's input value for this object (according to the simulated state) as the decided value.

  Recall that an s-agreement protocol may block forever if s or more processes fail in the middle of its execution. Thus, when it is used to simulate an access to (t, s)-set consensus, failures of s or more simulators may block t simulated processes.

  Also, recall that s-agreement may return different values to different simulators (as long as there are at most s of them). To ensure that the outcome of each of the t simulated processes accessing the (t, s)-set-consensus object is determined consistently by different simulators, the outcome of the simulated step is then agreed upon using 1-agreement.

If an agreement protocol for process p_ℓ blocks, simulator q_i proceeds to the next non-blocked simulated process in the round-robin order. If the corresponding agreement protocol terminates, the simulator updates the atomic-snapshot memory with its estimation of the simulated states of $p_1, \ldots, p_n$, where the new state of p_i is based on the outcome of the agreement.

Correctness. The use of 1-agreement protocols for both kinds of simulated operations implies that every step is simulated consistently, i.e., the simulators agree on the next simulated state of each process in $\{p_1, \ldots, p_n\}$.

The proposal to each of these agreement protocols is either the recently taken snapshot of the simulated system state (in case a snapshot operation is simulated) or the value that the simulated process must propose based on its state (in case an access to a (t, s)-set-consensus object is simulated). The initial state of each simulated process is an (agreed upon) input value of a participating simulator.

Each simulated snapshot is computed based on the most recent simulated states of $p_1, \ldots, p_n$ contained in the snapshot taken by the simulator "winning" the corresponding 1-agreement. The use of s-agreement in simulating accesses to a (t, s)-set-consensus object ensures that the simulated accesses return at most s proposed values. Thus, starting from the initial states of the simulated processes, we inductively derive that all states that appear in the simulated run are *compliant* with a run E of A: in E, each process p_i goes through the sequence of states that are agreed upon for p_i in the simulation.

Progress. It remains to show that at least one process in $\{p_1, \ldots, p_n\}$ *makes progress* in the simulated run, assuming that at least one of the $k+1$ simulators is correct. Consider any simulated run. We show that in this run, at least one of the simulated processes takes sufficiently many simulated steps (for producing an output for k-set consensus).

A simulated process may stop making progress only if an agreement protocol used for simulating its step blocks, which may happen only if a certain number of simulators stopped taking steps in the middle of the protocol.

Suppose that at most k simulators are faulty. Given that a faulty simulator can block at most one agreement protocol, we can identify the set of distinct agreement protocols $A_1 \ldots, A_p$ that are blocked in our run, and for all $j = 1, \ldots, p$, let A_i be s_i-agreement. We also identify p subsets of k faulty simulators of sizes $s_1, \ldots, s_p$, where s_i is the number of simulators that block A_i.

For each $i = 1, \ldots, p$, let t_i denote the number of simulated processes that are blocked because of A_i. If A_i is an instance of 1-agreement ($s_i = 1$) used to simulate a snapshot operation, to agree on the input of a given process, or to agree on the output of a set-consensus object at a given process, then we set $t_i = 1$ (only the corresponding simulated process can be blocked). Otherwise, A_i is an instance of s_i-agreement used to simulate an access to some (f_i, s_i)-set consensus, and we set $t_i = f_i$ (up to f_i processes accessing the (f_i, s_i)-set-consensus object can be blocked).

Since there are at most k faulty simulators, we get a multiset $\{(t_1, s_1), \ldots, (t_p, s_p)\}$ of elements in C such that $\sum_i s_i \leq k$. But then, by our contradiction hypothesis, we have $\sum_i t_i < n$, i.e., the total number of blocked simulated processes is less than n. Thus, at least one of the n processes $p_1, \ldots, p_n$ makes progress in the simulated run and eventually decides. Assuming that the first simulator to witness a decision in the simulated run writes it in the shared memory, we derive that every correct simulator eventually reads some decided value and decides.

Since all these values are coming from a run of an algorithm solving (n, k)-set consensus, there are at most k distinct decided values. Each of the decided values is an input of some simulator. Thus, $k+1$ simulators solve k-set-consensus using reads and writes – a contradiction. ◀

3.3 Computing the power of set-consensus collections

Having characterized the power of a collection C to solve set-consensus, we are now faced with the question of how to compute this power.

By Theorem 2, determining the best level of agreement that can be achieved by $C = \{(\ell_0, j_0), \ldots, (\ell_m, j_m)\}$ in a system of n processes is equivalent to finding $\min \sum_i j_i x_i$, under the constraints: $\sum_i \ell_i x_i \geq n$, $x_0, x_1, \ldots, x_m \in \{0, \ldots, n\}$. This can be viewed as a variation of the Knapsack optimization problem [21], where we aim at minimizing the total weight of a set of items from C put in a knapsack, while maintaing a predefined minimal total value of the knapsack content.[2] Here each j_i serves as the "weight" of an element in C, i.e., how much disagreement it may incur, and each ℓ_i serves as its "value", i.e., how many processes it is able to synchronize. We use this observation to derive an algorithm to compute AL_n^C in $O(n^2)$ steps.

Recall that C is represented as a monotonically increasing sequence $(\ell_0, j_0), \ldots, (\ell_m, j_m)$.

[2] The classical Knapsack optimization problem consists in maximizing the total value, while maintaining the total weight within a given bound.

First we *complete* C for the fixed system size n: for each $i = 1, \ldots, m$, such that $j_i < n$, we insert elements $(\max(j_i+1, \ell_{i-1}+1), j_i)$,$(\max(j_i+1, \ell_{i-1}+1)+1, j_i)$, ..., $(\min(\ell_i, n)-1, j_i)$, $(\min(\ell_i, n), j_i)$. For example, the completion of $C = \{(1,1),(3,2),(10,6)\}$ for $n = 11$ would be $\{(1,1), (3,2), (7,6), (8,6), (9,6), (10,6)\}$. Notice that since (ℓ_0, j_0), ..., (ℓ_m, j_m) is monotonically increasing, such a completion can be performed in $O(n)$ steps, and the resulting sequence is also monotonically increasing.

As a result of the completion, for every $r = 1, \ldots, n$, and each element of the kind $(\ell, j) \in C$ such that $\ell > r$ and $j < r$ we have a new element (r, j). As we will see below, this allows us to compute AL_r^C in $O(r^2)$ steps.

We observe that for all $r = 1, \ldots, n$, $AL_r^C = \min_{\ell_i \le r}(j_i + AL_{r-\ell_i}^C)$. Indeed, for all (ℓ_i, j_i) such that $\ell_i \le r$, it must hold that $j_i + AL_{r-\ell_i}^C \ge AL_r$, otherwise, (ℓ_i, j_i) plus the multiset $(t_1, s_1), \ldots, (t_p, s_p)$ of elements in C that reaches $AL_{r-\ell_i}$ would give $j_i + \sum_v s_v < AL_r^C$ and $\ell_i + \sum_v t_v \ge \ell_i + r - \ell_i = r$, contradicting the definition of AL_r^C. Further, since C is complete, for each multiset $(t_1, s_1), \ldots, (t_p, s_p)$ in C reaching AL_r^C, we can construct a multiset $(\min(t_1, r), t_1), \ldots, (\min(t_p, r), s)$ in C (each set-consensus object in the multiset is defined for at most r processes) that also reaches AL_r^C. Hence, $AL_r^C = \min_{\ell_i \le r}(j_i + AL_{r-\ell_i}^C)$

Thus, we can use the following simple iterative algorithm (a variant of a solution to the Knapsack optimization problem based on dynamic programming) to compute AL_n^C in $O(n^2)$ steps:

$AL_0^C = 0$;
for $r = 1, \ldots, n$ **do** $AL_r^C = \min_{\ell_i \le r}(j_i + AL_{r-\ell_i}^C)$.

In each iteration $r = 1, \ldots, n$ of the algorithm above, we perform at most r checks, which gives us $O(n^2)$ total complexity.

We can also consider a related notion of *j-set-consensus number* of C, denoted SCN_j^C and defined as the maximal number of processes that can achieve j-set consensus using C and read-write registers: $SCN_j^C = \max_{AL_n^C \le j} n$. This is a natural generalization of Herlihy's consensus power [18]. Note that the problem of computing SCN_j^C is the classical Knapsack optimization problem, and using a variation of the algorithm above we can do it in $O(j|C|)$ steps (see, e.g., [21, Chap. 5]).

4 An adaptive algorithm: reaching optimal agreement

Theorem 2 implies that, for every fixed n, there exists an AL_n^C-set-agreement algorithm $\mathcal{ST}_n^C$ ($\mathcal{ST}$ for *static*) using C. We show that these algorithms can be used in an *adaptive* manner, so that for each set of participating processes, the best possible level of agreement can be achieved.

To understand the difficulty of finding such an adaptive algorithm, consider $C = \{(1,1),$ $(13,5)$, $(20,9)\}$. For selected sizes of participating sets m, the table in Figure 1 gives AL_m^C, and lists the elements of C used in the corresponding $\mathcal{ST}_m^C$.

If we have 16 processes, $\mathcal{ST}_{16}^C$ uses one instance of $(13,5)$-set consensus and three instances of $(1,1)$-set consensus to achieve $AL_{16}^C = 9$. But if two new processes arrive we need $(20,9)$-set consensus to achieve $AL_{18}^C = 9$. Interestingly, to achieve $AL_{22}^C = 10$, we should abandon $(20,9)$-set consensus and use two instances of $(13,5)$-set consensus instead. In other words, we cannot simply add a set-consensus instance of the species we used before to account for the arrival of new processes. Instead, we have to introduce a new species.

We present a wait-free adaptive algorithm that ensures that if the set of participating processes is of size m, then at most AL_m^C distinct input values can be output. We call such an algorithm *optimally adaptive for C*.

m	AL^C_m	$\mathcal{ST}^C_m$
1	1	(1,1)
2	2	(1,1) (1,1)
3	3	(1,1) (1,1) (1,1)
4	4	(1,1) (1,1) (1,1) (1,1)
5 to 13	5	(13,5)
14	6	(13,5) (1,1)
15	7	(13,5) (1,1) (1,1)

m	AL^C_m	$\mathcal{ST}^C_m$
16	8	(13,5)(1,1)(1,1)(1,1)
17	9	(13,5)(1,1)(1,1)(1,1)(1,1) or (9,20)
18 to 20	9	(20,9)
21	10	(20,9)(20,9) or (13,5)(13,5)
22 to 26	10	(13,5) (13,5)
...		

Figure 1 Selecting elements in $C = \{(1,1), (13,5), (20,9)\}$ to solve AL^C_m-set consensus.

The algorithm is presented in Figure 2. The idea is the following: periodically, every process p writes its current value (initially, its input), together with the number of processes it has seen participating so far (initially, 0) in the shared memory, and takes a snapshot to get the current set P of participating processes and their inputs.

Process p then computes its *rank* in P and adopts the value v from a process announcing the largest participating set. The chosen input is then proposed to an instance of algorithm $\mathcal{ST}^C_{|P|}$, where p behaves as the process at position *rank* and proposes value v. More precisely, $\mathcal{ST}^C_{|P|}$ is treated as an algorithm for processes $q_1, \ldots, q_{|P|}$ and, thus, p runs the code of process q_{rank} in the algorithm with input value v.

Note that since the set is derived from an atomic snapshot of the memory, the notion of the largest participating set is well-defined: the snapshots of the same size are identical. Therefore, at most $|P|$ processes participate in $\mathcal{ST}^C_{|P|}$ and each of these $|P|$ processes can only participate at a distinct position corresponding to its rank in P. As a result, every correct process invoking $\mathcal{ST}^C_{|P|}$ will eventually get an output of the "best" set-consensus algorithm for P.

When the participating set P observed by p does not change in two consecutive iterations, p terminates with its current value.

▶ **Theorem 3.** *Let $\mathcal{C}$ be a set-consensus collection, n be an integer. The algorithm in Figure 2 is optimally adaptive for $\mathcal{C}$ in a system of n processes.*

Proof. We show first that every correct process eventually returns a value, and any returned value is a proposed one.

Let p and q take snapshots (Lines 2 or 11) in that order, and let P_p and P_q be, respectively, the returned participating sets. We observe first that $P_p \subseteq P_q$. Indeed, each position in the snapshot object R is initialized to $(\perp, \perp)$. Once, p updates $R[p]$ with its value and participation level, the position remains non-$\perp$ forever. Thus, if q takes its snapshot of R after p, then P_p, the set of processes whose positions are non-$\perp$ in the resulting vector, is a subset of P_q.

Therefore, the sets P and *parts* evaluated by p in Line 11 are non-decreasing with time. Since $\mathcal{ST}^C_{|parts|}$ is wait-free, the only reason for a correct process p not to return is to find that $parts \subsetneq P$ in Line 13 infinitely often, i.e., both P and *parts* grow indefinitely. But the two sets are bounded by the set Π of all processes – a contradiction.

Furthermore, every returned value is a value decided in an instance of $\mathcal{ST}^C_{|parts|}$. But every value proposed to algorithm $\mathcal{ST}^C_{|parts|}$ was previously read in a non-$(\perp, \perp)$ position of R, which can only contain an input value of some process.

Hence, every correct process eventually returns a value, and any returned value is a proposed one.

Shared objects:

R: `snapshot object, storing pairs` $(value, level)$, `initialized to` $(\bot, \bot)$

Local variables for process $p \in \Pi$*:*

$r[1, \ldots, n]$: `array of pairs` $(value, level)$
$prop, v$: *value*
$parts, P \in 2^{\Pi}$
$index$: integer

Code for process $p \in \Pi$ *with proposal* v_p*:*

1 $R.update(p, (v_p, 0))$
2 $r[1, \ldots, n] = R.snapshot()$
3 P=set of processes q such that $r[q] \neq (\bot, \bot)$
4 **repeat**
5 $\quad parts := P$
6 $\quad rank :=$ the rank of p in *parts*
7 $\quad k :=$ be the greatest integer such that $(-, k)$ is in r
8 $\quad v :=$ be any value such that (v, k) is in r
9 $\quad prop := \mathcal{ST}^{C}_{|parts|}$ with value v at position *rank*
10 $\quad R.update(p, (prop, |parts|))$
11 $\quad r[1, \ldots, n] = R.snapshot()$
12 $\quad P$=set of processes q such that $r[q] \neq (\bot, \bot)$
13 **until** $parts = P$
14 **return** *prop*

Figure 2 An optimally adaptive set-consensus algorithm.

Now consider a run of the algorithm in Figure 2 in which m processes participate. We say that a process p *returns at level* t in this run if it outputs (in Line 14) the value *prop* returned by the preceding invocation of $\mathcal{ST}^{C}_{t}$ (in Line 9). By the algorithm, if p returns at level t, then the set *parts* of processes it witnessed participating is of size t.

Let ℓ be the smallest level ($1 \leq \ell \leq n$) at which some process returns, and let O_ℓ be the set of values ever written in R at level ℓ, i.e., all values v, such that (v, ℓ) appears in R.

We show first that for all $\ell' > \ell$, if R contains (v', ℓ'), then $v' \in O_\ell$.

By contradiction, suppose that some process q is the first process to write a value (v', ℓ') (in Line 10), such that $\ell' > \ell$ and $v' \notin O_\ell$, in R. Thus, the immediately preceding snapshot taken by q before this write (in Lines 2 or 11) witnessed a participating set of size ℓ'. Hence, the snapshot of q occurs after the last snapshot (of size $\ell < \ell'$) taken by any process p that returned at level ℓ. But immediately before taking its last snapshot, every such process p has written (v, ℓ) in R (Line 10) for some $v \in O_\ell$. Thus q must see (v, ℓ) in its snapshot of size ℓ' and, since, by the assumption, the snapshot contains no values written at levels higher than ℓ, q must adopt some value written at level ℓ (Lines 7 and 8). Thus, $v' \in O_\ell$ – a contradiction.

Thus, every returned value must appear in O_ℓ, where ℓ is the smallest level ($1 \leq \ell \leq n$) at which some process returns. Now we show that $|O_\ell| \leq AL^C_m$, recall that m is the number of participating processes.

Indeed, since all values that appear in O_ℓ were previously returned by the algorithm $\mathcal{ST}^C_\ell$ (Line 9) and, as we observed earlier, the algorithm is used by at most ℓ processes, each choosing a unique position based on its rank in the corresponding snapshot of size ℓ, there can be at most AL^C_ℓ such values. Since at most m processes participate in the considered run, we have $\ell \leq m$, and, thus, $AL^C_\ell \leq AL^C_m$.

Hence, in a run with participating set of size m, $|O_\ell| \leq AL^C_m$ and, thus, at most AL^C_m values can be returned by the algorithm. Thus, we indeed have an optimally adaptive set-consensus algorithm using C. ◀

On unbounded concurrency. Our definitions of the agreement level and the set-consensus number of a set-consensus collection are independent of the size of the system: they are defined with respect to a given participation level. Our adaptive algorithm does account for the system size, as it uses atomic snapshots. But by employing the atomic-snapshot algorithms for unbounded-concurrency models described in [16], we can easily extend our adaptive solution to these models too.

5 Related work

Our algorithm computing the power of a set-consensus collection in $O(n^2)$ steps (for a system of n processes) is inspired by the dynamic programming solution to the Knapsack optimization problem described, e.g., in [21, Chap. 5].

Herlihy [18] introduced the notion of *consensus number* of a given object type, i.e., the maximum number of processes that can solve consensus using instances of the type and read-write registers. It has been shown that n-process consensus objects have consensus power n. However, the corresponding consensus hierarchy is in general not robust, i.e., there exist object types, each of consensus number 1 which, combined together, can be used to solve 2-process consensus [22]. Besides objects of the same consensus number m may not be equivalent in a system of more than m processes [2].

Borowsky and Gafni [4], and then Chaudhuri and Reiners [8, 23] independently explored the power of having multiple instances of (ℓ, j)-set-consensus objects in a system of n processes with respect to solving set consensus, which is a special case of the question considered in this paper. The characterization of [4, 8, 23] is established by a generalized BG simulation [3, 5] by Borowsky and Gafni, where instead of 1-agreement protocol, a more general j-agreement protocol is used. Our results employ this agreement protocol to show a more general result.

Gafni and Koutsoupias [13] and Herlihy and Rajsbaum [19] showed that wait-free solvability of tasks for 3 or more processes using registers is an undecidable question. We show that in a special case of solving set consensus using a set-consensus collection, the question is decidable. Moreover, we give an explicit polynomial algorithm for computing the power of a set-consensus collection.

6 Concluding remarks

We hope that this work will be a step towards proving a more general conjecture that our set-consensus numbers capture precisely the computing power of any "natural" shared-memory model. An indication that the conjecture is true is that set-consensus objects are, in a precise sense, *universal* (generalizing the consensus universality [18]): using (n,k)-set-consensus objects, n processes can implement k independent sequential state machines so that at least one of them is able to make progress, i.e., to execute infinitely many commands [12]. Popular restrictions of the runs of the wait-free model, such as *adversaries* [10] and *failure detectors* [7, 6], were successfully characterized via their power for solving set consensus [14, 15, 9]. Also, it can be inferred from the recent result by Afek et al. [2] that, like consensus, set-consensus objects can express precisely certain deterministic objects [2]. We therefore believe that the power of a large class of "natural" models (determined by restrictions mentioned above) can be captured by their ability to solve set consensus. This class must exclude models in which "in between" objects, like *Weak Symmetry-Breaking* [20, 17], are used: such models, as we believe, cannot be expressed as a restriction of the runs of the wait-free model, and are therefore not "natural".

References

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

2 Yehuda Afek, Faith Ellen, and Eli Gafni. Deterministic objects: Life beyond consensus. In *PODC*, 2016.

3 Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *STOC*, pages 91–100. ACM Press, May 1993.

4 Elizabeth Borowsky and Eli Gafni. The implication of the Borowsky-Gafni simulation on the set-consensus hierarchy. Technical report, UCLA, 1993. `http://fmdb.cs.ucla.edu/Treports/930021.pdf`.

5 Elizabeth Borowsky, Eli Gafni, Nancy A. Lynch, and Sergio Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14(3):127–146, 2001.

6 Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

7 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

8 Soma Chaudhuri and Paul Reiners. Understanding the set consensus partial order using the Borowsky-Gafni simulation (extended abstract). In *WDAG*, pages 362–379, 1996.

9 Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Petr Kuznetsov. Wait-freedom with advice. *Distributed Computing*, 28(1):3–19, 2015.

10 Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Andreas Tielmann. The disagreement power of an adversary. *Distributed Computing*, 24(3-4):137–147, 2011.

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

12 Eli Gafni and Rachid Guerraoui. Generalized universality. In *Proceedings of the 22nd international conference on Concurrency theory*, CONCUR'11, pages 17–27, Berlin, Heidelberg, 2011. Springer-Verlag.

13 Eli Gafni and Elias Koutsoupias. Three-processor tasks are undecidable. *SIAM J. Comput.*, 28(3):970–983, 1999.

14 Eli Gafni and Petr Kuznetsov. Turning adversaries into friends: Simplified, made constructive, and extended. In *OPODIS*, pages 380–394, 2010.

15 Eli Gafni and Petr Kuznetsov. Relating L-Resilience and Wait-Freedom via Hitting Sets. In *ICDCN*, pages 191–202, 2011.

16 Eli Gafni, Michael Merritt, and Gadi Taubenfeld. The concurrency hierarchy, and algorithms for unbounded concurrency. In *PODC*, pages 161–169, 2001.

17 Eli Gafni, Sergio Rajsbaum, and Maurice Herlihy. Subconsensus tasks: Renaming is weaker than set agreement. In *DISC*, pages 329–338, 2006.

18 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, January 1991.

19 Maurice Herlihy and Sergio Rajsbaum. The decidability of distributed decision tasks (extended abstract). In *STOC*, pages 589–598, 1997.

20 Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(2):858–923, 1999.

21 Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack problems*. Springer, 2004.

22 Wai-Kau Lo and Vassos Hadzilacos. All of us are smarter than any of us: Nondeterministic wait-free hierarchies are not robust. *SIAM J. Comput.*, 30(3):689–728, 2000.

23 Paul Reiners. Understanding the set consensus partial order using the Borowsky-Gafni simulation. Master's thesis, Iowa State University, 1996.

24 Michael Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM J. on Computing*, 29:1449–1483, 2000.

A An ℓ-agreement algorithm

The algorithm (presented in Figure 3) uses two *atomic snapshot* objects A and B, initialized with $\perp$'s. A process writes its input in A (line 15) and takes a snapshot of A (line 16). Then the process writes the outcome of the snapshot in B (line 17) and keeps taking snapshots of B until it finds that at most $\ell-1$ participating (i.e., having written their values in A) processes that have not finished the protocol, i.e., have not written their values in B (Lines 18-21). Finally, the process returns the smallest value (we assume that the value set is ordered) in the smallest-size non-$\perp$ snapshot found in B (containing the smallest number of non-$\perp$ values). (Recall that all snapshot outcomes are related by containment, so there indeed exists such a smallest snapshot.)

▶ **Theorem 4.** *The algorithm in Figure 3 implements ℓ-agreement.*

Proof. The validity property (i) is immediate: only the identifier of a participating process can be found in a snapshot object. The termination property (iii)′ of ℓ-agreement is immediate:

Shared objects:
  A, B: `snapshot objects, initially` $\perp$

propose(v)
15 A.*update*(v)
16 U := A.*snapshot()*
17 B.*update*(U)
18 **repeat**
19   W := B.*snapshot()*
20   $X := \{j | (U[j] \neq \perp) \wedge (W[j] = \perp)\}$
21 **until** $|X| \leq \ell - 1$
22 S := the smallest-size set of non-$\perp$ values contained in $\{ W[j];\ j = 1, \ldots, n,\ W[j] \neq \perp\}$
23 *return* $\min(S)$

Figure 3 The ℓ-agreement algorithm.

if at most $\ell - 1$ processes that have executed line 15 fail to execute line 17, then the exit condition of the repeat-until clause in line 21 eventually holds and every correct participating process terminates.

Suppose, by contradiction, that at least $\ell+1$ different values are returned by the algorithm. Thus, at least $\ell + 1$ distinct snapshots were written in B by $\ell + 1$ processes. Let L be the set of processes that have written the ℓ smallest snapshots in B in the run. The set is well-defined as all snapshots taken in A are related by containment. We are going to establish a contradiction by showing that every process must return the smallest value in one of the snapshots written by the processes in L and, thus, at most ℓ distinct inputs will be produced.

Let p_i be any process that completed line 17 by writing the result of its snapshot of A in B. Let U be the set of processes that p_i witnessed in A and, thus, wrote to its position in B in line 17.

If $p_i \in L$, i.e., U is one of the ℓ smallest snapshots ever written in B, then p_i will return the value of the smallest process in U or a smaller snapshot written by some process in L. If $p_i \notin L$, then U contains all ℓ distinct snapshots written by the processes in L. Since each process in L is included in the snapshot it has written in B, we derive that $L \subseteq U$. Since p_i returns a value only if all but at most $\ell - 1$ processes it witnessed participating have written their snapshots in B, at least one snapshot written by a process in L is read by p_i in B. Thus, p_i outputs the value of the smallest process in the snapshot written by a process in L – a contradiction.

Thus, at most ℓ distinct values can be output and (ii)$'$ is satisfied. ◀