Task Computability in Unreliable Anonymous Networks

Petr Kuznetsov
LTCI, Télécom ParisTec, Université Paris-Saclay, France
petr.kuznetsov@telecom-paristech.fr

Nayuta Yanagisawa
DeNA Co., Ltd., Japan
nayuta.yanagisawa@dena.com

Abstract

We consider the *anonymous broadcast* model: a set of n anonymous processes communicate via *send-to-all* primitives. We assume that underlying communication channels are asynchronous but reliable, and that the processes are subject to *crash* failures. We show first that in this model, even a single faulty process precludes implementations of *atomic* objects with non-commuting operations, even as simple as read-write registers or add-only sets. We, however, show that a *sequentially consistent* read-write memory and add-only sets can be implemented t*-resiliently* for $t < n/2$, i.e., provided that a majority of the processes do not fail. We use this implementation to establish an equivalence between the t-resilient read-write anonymous shared-memory model and the t-resilient anonymous broadcast model in terms of colorless task solvability. As a result, we obtain the first task computability characterization for *unreliable* anonymous message-passing systems.

2012 ACM Subject Classification Theory of computation → Distributed computing models, Theory of computation → Computability

Keywords and phrases Distributed tasks, anonymous broadcast, fault-tolerance

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.23

1 Introduction

Algorithms for conventional distributed systems assume that every process is assigned a distinct identifier that can be used in the code. However, *anonymous* algorithms that have to program processes identically can be used in a much wider context. Many systems, such as sensor networks, web services, peer-to-peer file repositories, are anonymous by design or may choose anonymity for the sake of users' privacy [2, 10].

In this paper, we consider computability issues of anonymous systems. This topic is traditionally tackled in *shared-memory* models where a set of asynchronous (or partially synchronous) anonymous processes communicate via *multi-writer multi-reader* shared memory locations. A number of interesting complexity and computability bounds [16, 7, 12, 20, 8, 6] have been established for these models. Guerraoui and Ruppert [16] showed that many interesting abstractions, such as timestamps and atomic snapshots, can be implemented in a *wait-free* way, i.e., tolerating an arbitrary number of faulty processes. Yanagisawa [20] characterized the class of *colorless tasks* that can be solved wait-free in this model.[1] It turns

[1] Informally, a colorless task, such as consensus or set agreement, is defined in terms of relations between *sets* of inputs and *sets* of outputs, so they allow natural formulations for anonymous systems. A wait-free algorithm guarantees that a process *makes progress*, e.g., produces a task output, in a finite number of its own steps, even if all $n-1$ remaining processes are faulty.

© Petr Kuznetsov and Nayuta Yanagisawa;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 23; pp. 23:1–23:13

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

out the class is precisely the tasks that can be solved in the conventional *non-anonymous* model, i.e., getting rid of process identifiers does not make the system weaker with respect to solving colorless tasks. The equivalence has been recently generalized to t-resilient models assuming that at most t processes are allowed to fail by Delporte et al. [13]: a colorless task is t-resiliently solvable in the (non-anonymous) read-write shared-memory model if and only it is t-resiliently solvable in the anonymous counterpart.

Anonymous computing in networks. One can argue that shared memory may be an inadequate communication model for *large-scale* systems, which typically motivate assuming anonymous algorithms. In this paper, we propose to have a closer look at anonymous computations in *message-passing* systems. More precisely, we consider systems in which processes communicate via the reliable *anonymous broadcast* primitive: every broadcast message is eventually received and the integrity of received messages is ensured, while it is impossible to detect the source of a given message.

The model has been considered earlier in the *fault-free* context by Aspnes et al. [3]. They showed that if no process can fail, any *idemdicent* object can be implemented in the anonymous broadcast model in the *linearizable* way, i.e., ensuring that each operation on it appears to be executed atomically within the operation's interval. Intuitively, for an idemdicent object, such as a *register*, an *add-only set*, or a *counter* with separate increment and read operations, any operation modifying the object's state returns only a (non-informative) *ack* response.

Unreliable anonymous networks. We propose a set of computability results for the anonymous broadcast model with *unreliable* processes. We show first that no object exporting non-commuting operations, including registers and add-only sets, has a linearizable implementation in this model, as long as just a single process can fail. We leverage the fact that, in anonymous message-passing systems, no process can detect whether a given message comes in reaction to its own action or to an earlier action performed by its *clone*, i.e., a process having an identical state. Therefore, it might become impossible to ensure that operations on the implemented object appear in an order that preserves their real-time precedence, which is required by linearizability.

We show, however, that a t-resilient *sequentially consistent* add-only set can be implemented in the anonymous broadcast model, assuming that $t < n/2$, i.e., less than half of the processes are allowed to fail. Unlike linearizable objects, sequentially consistent ones do not ensure that the sequential execution preserves the real time precedence between operations and, as a result, sequential consistency is not a *composable* property [17]. Our t-resilient implementation of an add-only set, inspired by the wait-free atomic snapshot algorithm by Afek et al. [1], is interesting in its own right. To ensure that the views of the set contents evaluated by different processes are consistent with the same sequential execution, a *get* operation is only allowed to return a set of values that is agreed upon by a majority of processes. To guarantee that every correct process completes each of its operations, we introduce a *helping* mechanism: every value added to the set is equipped with a view that can be used by future *get* operations.

Colorless tasks in unreliable anonymous networks. Further, we show that a sequentially consistent add-only set is as good as a linearizable one, as long as colorless task computability is concerned. More precisely, we can characterize the class of colorless tasks that are t-resiliently solvable in the anonymous broadcast model, where $t < \frac{n}{2}$. The characterization

stipulates that a colorless task is t-resilient solvable in the *anonymous broadcast* model if and only if it is t-resiliently solvable in the anonymous *shared-memory* model, where $t < \frac{n}{2}$. Thus, for colorless tasks, the anonymous broadcast model with a majority of correct processes is as powerful as the anonymous shared-memory model and, by recalling the equivalence of Delporte et al. [13], the regular (non-anonymous) shared-memory model.

Technically, the *if* part of our characterization is established by the following two steps: First, we present an implementation of a *sequentially-consistent* snapshot. Then, we show that any algorithm that t-resiliently solves a colorless task using an atomic snapshot memory also solves the colorless task with a sequentially-consistent snapshot memory. The *only if* part follows from the characterization of the t-resilient colorless task solvability in the anonymous *shared-memory* model by Delporte et al. [13].

Related work. Colorless tasks, a fundamental class of distributed problems, include consensus [15], set agreement [9], and loop agreement [19]. The class was extensively studied mainly in the context of the non-anonymous shared-memory computing [18]. There are recent papers that have studied the computability of colorless tasks in the anonymous shared-memory model [20, 13].

In the non-anonymous setting, the message-passing model and the shared-memory model are equivalent in the sense of simulation [4], where $t < \frac{n}{2}$. Thus, a colorless task can be solved in the shared-memory model if and only if it can be solved in the t-resilient message-passing model. The situation is different in the anonymous case. In general, the anonymous broadcast model and the anonymous shared-memory model cannot simulate each other. Thus, results concerning the anonymous shared-memory colorless task computability cannot be directly applied to the anonymous broadcast model.

The add-only set object has been first introduced under the name of *weak set* by Delporte and Fauconnier [11]. The object has been studied mainly in the context of anonymous shared-memory computing and used to characterize anonymous shared-memory systems concerning colorless tasks [20, 13, 14].

Failure detectors that enable solutions to the consensus task in unreliable anonymous networks have been discussed by Bonnet and Raynal [5]. In contrast, this paper focuses on general task computability in asynchronous anonymous networks, i.e., without help from external oracles.

Roadmap. The rest of the paper is organized as follows. In Section 2, we give our model definitions. In Section 3, we show that no type with weakly non-commutative operations allows a linearizable implementation in an unreliable atomic broadcast model. In Section 4, we present a sequentially consistent implementation of an add-only set in the anonymous broadcast model with a majority of correct processes. In Section 5, we present our anonymous colorless task computability theorem. In Section 6, we discuss relaxations of linearizability that might fit the anonymous broadcast context.

2 Preliminaries

We consider the *anonymous* model of n processes, $p_1, \ldots, p_n$, that have no knowledge of their identifiers and execute an identical algorithm. We particularly focus on the *anonymous asynchronous broadcast model*. In the model, a distributed system is composed of n anonymous processes that run asynchronously and communicate by broadcasting messages via a fully connected reliable network: every pair of processes are connected via a first-in-first-out

(FIFO) reliable link. To send a message m to all, a process invokes *broadcast*(m) primitive, and an event *receive*(m) occurs when the message is received.

Broadcast models. Here we distinguish two broadcast models: *instantaneous* and *non-instantaneous* broadcasts. In the instantaneous broadcast model, every event *broadcast*(m) performed by a process p_i instantaneously sends message m to every other process and, as the links are assumed to be reliable, it is guaranteed that the message will eventually be received by every correct process. In the non-instantaneous broadcast model, if *broadcast*(m) is the last event performed by a *faulty* process, then it sends m to a *subset* of processes and guarantees that the message will be eventually received by every correct process *in this subset*. In both cases, the communication channels ensure FIFO semantics: the messages are delivered in the order they have been sent. Notice, however, that a process cannot detect through which link a message has been received.

Our impossibility result is shown for the stronger instantaneous model and our upper bounds hold for the weaker non-instantaneous model. (By default we assume that the model is non-instantaneous.)

Object types and implementations. A sequential object type is defined as a tuple $T = (Q, q_0, O, R, \Delta)$, where Q is a set of states, $q_0 \in Q$ is an initial state, O is a set of operations, R is a set responses and $\Delta \subseteq Q \times O \times Q \times R$ is a relation that associates a state and an operation to a set of possible new states and corresponding responses. Here we assume that Δ is total on the first two elements, i.e., for each state $q \in Q$ and each operation in $o \in O$, some transition to a new state is defined, i.e., $\forall (q, o) \in Q \times O, \exists (q', r) \in Q \times R$: $(q, o, q', r) \in \Delta$.

A *history* is a sequence of (operation) invocations and responses and a sequential history is a history that starts with an invocation of an operation and in which every invocation is immediately followed with a *matching* response. A *sequential history* $o_1, r_1, o_2, r_2, \ldots$, where $\forall i \geq 1, o_i \in O,\ r_i \in R$, is *legal* with respect to type $T = (Q, q_0, O, R, \Delta)$ if there exists a sequence $q_1, q_2, \ldots$ of states in Q such that $\forall i \geq 1$, $(q_{i-1}, o_i, q_i, r_i) \in \Delta$.

An *implementation* of an object type T is an algorithm that, for each invoked operation, prescribes the actions that a process needs to take to perform it. In our case, the actions are invocations of *broadcast* primitives, processing of *receive* events and returning responses to the invoked operations. An *execution* of an implementation is a sequence of *events*: invocations and responses of operations, *broadcast* calls and *receive* events: the sequence of events at every process must respect the algorithm assigned to it. A process is called *faulty* in an infinite execution if it stops before performing an event prescribed by its algorithm; otherwise it is called *correct*.

Linearizability and sequential consistency. We assume that no process invokes a new operation before receiving a response for the previous one. For each pattern of invocations, the implementation produces a *history*, i.e., the sequence of distinct invocations and responses, labelled with process identifiers and unique sequence numbers.

A projection of a history H to process p_i, denoted $H|i$ is the subsequence of elements of H labelled with p_i. An invocation o by a process p_i is *incomplete* in H if it is not followed by a response in $H|i$. A history is *complete* if it has no incomplete invocations. A *completion* of H is a history $\bar{H}$ that is identical to H except that every incomplete invocation in H is either removed or *completed* by inserting a matching response somewhere after it.

A *sequentially consistent* implementation of an object type T ensures that for every history H it produces, there exists a completion $\bar{H}$ and a legal sequential history S such that for all processes p_i, $\bar{H}|i = S|i$.

A *linearizable* implementation, additionally, preserves the real-time order between invocations. Formally, an invocation o_1 *precedes* an invocation o_2 in H, denoted $o_1 \prec_H o_2$, if o_1 is complete and the corresponding response r_1 precedes o_2 in H. Note that $\prec_H$ stipulates a partial order on invocations in H. Now a linearizable implementation of T ensures that for every history H it produces, there exists a completion $\bar{H}$ and a legal sequential history S such that (1) for all processes p_i, $\bar{H}|i = S|i$ and (2) $\prec_H \subseteq \prec_S$.

A (sequentially consistent or linearizable) implementation is *t-resilient* if, under the assumption that at most t processes crash, it ensures that every invocation performed by a correct process is eventually followed by a response.

An *anonymous* implementation is not allowed to use process identifiers: every process is assigned the same algorithm that only depends on the sequence of operation invocations and received messages.

3 Linearizable objects with non-commuting operations

In this section, we show that nontrivial sequential object types cannot be implemented in a linearizable way in the anonymous broadcast model if $t \geq 1$, i.e., at least one process may fail.

Given a type $T = (Q, q_0, O, R, \Delta)$, we say that operations $o_1, o_2 \in O$ are *weakly-non-commutative* if, for all $r_1, r_2 \in R$ such that $o_1 r_1 o_2 r_2$ is legal, $o_1 r_1 o_2 r_2 o_1 r_1$ is not legal. Intuitively, o_1 is a read operation and o_2 is an update: swapping the order of the two operations affects the response of o_1.

Two examples of types with weakly non-commutative operations are *read-write register* and *add-only set*. A register stores an integer value, initially 0, and exports operations *read*, which returns the value, and *write*(v), $v \in \mathbb{N}$, which replaces the value with v. Here *read*() and *write*(1) are examples of weakly non-commutative operations. An add-only set stores a set of integer values, initially $\emptyset$, and exports operations *get*, which returns the set, and *add*(v), $v \in \mathbb{N}$, which adds v to the set. Similarly, *get*() and *add*(1) are weakly non-commutative.

▶ **Theorem 1.** *There does not exist a* 1*-resilient linearizable implementation of a type with weakly non-commutative operations in the anonymous instantaneous broadcast model.*

Proof. Suppose, by contradiction, that there exists such an implementation of a type $T = (Q, q_0, O, R, \Delta)$ with weakly non-commutative operations. Let $o_1, o_2 \in O$ be operations of T such that for all $r_1, r_2 \in R$, whenever $o_1 r_1 o_2 r_2$ is legal, it holds that $o_1 r_1 o_2 r_2 o_1 r_1$ is not legal.

Consider the following execution of the implementation. Process p_1 invokes o_1 and takes steps of the algorithm until a response r_1 is returned, then invokes o_2 and takes steps of the algorithm until a response r_2 is returned. In this execution, we pick another process p_2 and delay all messages broadcast *by and to* p_2 at least until r_2 is returned at p_1. The remaining processes obediently take steps of the algorithm and all messages sent across these processes are eventually received. Note that both o_1 and o_2 must eventually return at p_1, as the implementation is 1-resilient and, from p_1's perspective, the execution is indistinguishable from an execution in which p_2 has crashed initially and all other processes are correct. Let α be the resulting (finite) execution and $e_1, \ldots, e_k$ be the subsequence of events of p_1 in α that starts with o_1 and ends with r_1.

Let $e'_1, \ldots, e'_k$ be the sequence of events that is identical to $e_1, \ldots, e_k$, except that every event is now labelled with p_2 (or, in other words, is performed by p_2).

We claim that $\alpha e'_1, \ldots, e'_k$ is an execution of our implementation. By the construction of α, e'_1 is the invocation of o_1, i.e., $\alpha e'_1$ is an execution of the implementation. Inductively,

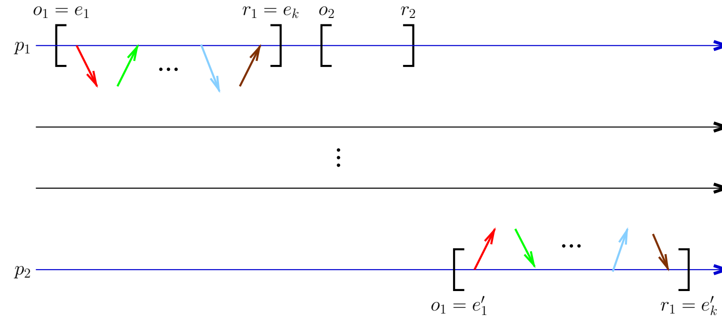

Figure 1 The execution constructed in the proof of Theorem 1. Processes p_1 and p_2 are clones: they execute exactly the same sequence of events in which operation o_1 returns response r_1.

suppose that for some ℓ, $1 \leq \ell < k$, $\alpha e'_1, \ldots, e'_\ell$ is an execution of the implementation. Note that, since the algorithm is anonymous, the local state of p_2 after $\alpha e'_1, \ldots, e'_\ell$ is identical to the local state of p_1 after $e_1, \ldots, e_\ell$.

Thus, if $e_{\ell+1}$ is a broadcast event, then p_2 can also perform a similar broadcast event $e'_{\ell+1}$ after $\alpha e'_1, \ldots, e'_\ell$ and, thus, $e'_{\ell+1}$ after $\alpha e'_1, \ldots, e'_{\ell+1}$ is indeed an execution.

Now suppose that $e_{\ell+1}$ is a receive event, for some message m previously broadcast by a process p_i. Recall that in α, p_2 has not received a single broadcast message. Note that in $\alpha e'_1, \ldots, e'_\ell$, p_2 received every message received by p_1 in $e_1, \ldots, e_\ell$, but no other messages. Thus, as m is the next message to be received by p_1 from p_i in $e_1, \ldots, e_\ell$, it is also the next message to be received by p_2 from p_i in $\alpha e'_1, \ldots, e'_\ell$. Hence, $\alpha e'_1, \ldots, e'_{\ell+1}$ is indeed an execution of our implementation. By induction, we constructed an execution $\alpha e'_1, \ldots, e'_k$ which produces a history $o_1 r_1 o_2 r_2 o_1 r_1$ that is not legal – a contradiction. ◀

4 Sequentially consistent add-only set

We now describe a sequentially consistent t-resilient implementation of *add-only set* in the anonymous broadcast model.

Overview. In spirit, our t-resilient implementation (Algorithm 1) is close to the wait-free atomic snapshot implementation by Afek et al. [1]. Similar to the snapshot operation in the algorithm by Afek al., a *get* operation executed by p_i repeatedly broadcasts its current (constantly evolving) evaluation of the set contents until either "sufficiently many" processes agree with it, or some process p_j reports a set of values containing all the values known to p_i at the beginning of the operation (in which case p_j helps p_i with completing the operation).

The algorithm proceeds in monotonically increasing *rounds*. Every process p_i maintains a set of values V_i, its local version of the set contents, where each value is equipped with a "witness set" (the set obtained by a *get* operation before the value has been added).

A *get* operation returns a set of values as soon as it establishes that a majority of processes evaluate the same set of values *for the current round* (line 21). Here, given a set of tuples $[u, W]$, where u is a value and W is a set of values "witnessed" by the process that added u, *values*(V) returns the set of first entries of tuples in V.

Otherwise, the process updates its local set with new values and proceeds to the next round. Note that if no values are added to the set from some point on, every get operation

Algorithm 1 Implementation of sequentially consistent add-only set: code for process p_i

```
1  Local variables:
2      V_i: set initially ∅        // local estimate of the set of values,
3                                  // equipped with witness sets
4      M_i^r:multiset initially ∅  // messages of round r
5      r_i: integer initially 0    // current round number

6  add(v):
7      Vals = get()
8      V_i = V_i ∪ {[v, Vals]}
9      broadcast([V_i])

10 upon receive [V]
11     V_i = V_i ∪ V

12 get():
13     U = V_i
14     while true do
15         r_i++
16         V = V_i
17         if M_i^{r_i} = ∅ then
18             broadcast([V, r_i])
19         wait until |M_i^{r_i}| > n/2
20         if ∀V' ∈ M_i^{r_i} : V' = V then
21             return values(V)
22         if ∃V' ∈ M_i^{r_i}, ∃[u, W] ∈ V' : values(U) ⊆ W then
23             return W

24 upon receive [V, r]
25     V_i = V_i ∪ V
26     if M_i^r = ∅ then
27         broadcast([V_i, r])
28     M_i^r = M_i^r ∪ {V}
```

invoked by a correct process will eventually return. Thus, the only reason for a get operation not to return is an infinite number of successful add operations performed by a concurrent process. To ensure that each correct process eventually completes every operation it invokes, we introduce a *helping* mechanism: before adding a new value to the set, a process *gets* the current set of values (line 7) and attaches it to the new value. If, within a *get* operation, a process finds a value equipped with a "sufficiently recent" outcome of another *get* operation, it adopts it and outputs as its own (line 23).

Of course, in an anonymous system, detecting the source of a received message is impossible. Therefore, to guarantee that all received messages associated with a round r are coming from distinct processes and to render the "majority condition" in line 19 meaningful, we require that a process broadcasts a message associated with a given round exactly once (lines 25-27). Of course, such messages can be associated to an identical request sent by a process different from p_i (clone of p_i). But, as we will see below, this is not an issue for the algorithm's correctness. To account for a "slow" process that might receive messages associated with rounds the process has not reached yet, it proactively stores all the messages it receives.

Correctness. Fix an execution of Algorithm 1. We show first that *views*, sets of values returned by *get* operations, are ordered by containment.

▶ **Lemma 2.** *For any two views U and W returned by get operations, $U \subseteq W$ or $W \subseteq U$.*

Proof. Note that each set of values U returned by a *get* operation is either evaluated by the operation itself and returned in line 21 or "adopted" from another *get* operation that terminated earlier and returned in line 23. But there can be only finitely many *get* operations that terminated before a given *get* operation returns. Thus, each such view U was returned in line 21 by some *get* operation. We then can associate U with the round number r in which this happened, we say U *was returned* in round r.

Let U and W be returned, respectively in rounds r and r', $r \leq r'$. We show below that $U \subseteq W$. Indeed, a *get* operation returns U in round r if a majority of processes broadcast the same message $[V, r]$ such that $values(V) = U$. Since a process only broadcasts its set for a given round only once, two processes returning in a given round return the same set of values. Furthermore, if a process returns U in round r, then every process p_i that passed through that round, will get $U \subseteq values(V_i)$. Thus, W returned in round $r' \geq r$ must satisfy $U \subseteq W$. ◀

▶ **Lemma 3.** *If two* get *operations invoked by the same process p_i return U and then W, then $U \subseteq W$.*

Proof. By the algorithm, every view returned by a *get* operations (lines 13 and 22) invoked by a process p_i contains $values(V_i)$ at the time of the invocation. As V_i grows monotonically with time (lines 8 and 25), we have $U \subseteq W$. ◀

▶ **Lemma 4.** *If p_i complete an $add(v)$ and later a get operation that returns U, then $v \in U$.*

Proof. As every *get* operation by p_i returns a superset of $values(V_i)$ evaluated at the moment of its invocation (line 13), and every $add(v)$ operation by p_i adds v to $values(V_i)$ (line 13), and V_i grows monotonically with time (lines 8 and 25), we have $v \in U$. ◀

▶ **Lemma 5.** *Every operation invoked by a correct process eventually returns.*

Proof. Suppose, by contradiction, that an operation *op* invoked by a correct process p_i never terminates. Note that *op* must be a *get* operation, either invoked directly or within an *add* operation. By the algorithm, as at most $t < n/2$ processes are faulty, p_i goes through infinitely many rounds in lines 14–23. Let U be the set of values (line 13) with which p_i started *op*.

Suppose first that there is a time after which no process completes an *add* operation. Then, there is a time after which no process p_j adds a new value to its local V_j. Since the broadcast channels are reliable, every message sent by a correct process is eventually received by every correct process (line 27). Thus, there exists a round r and a set of tuples V such that every process that broadcasts a message of the kind $[V', r']$, where $r' \geq r$, will have $V' = V$. Eventually, after sufficiently many rounds, p_i will reach a round for which every received set of values is V and, thus, p_i will return $values(V)$ in line 23 – a contradiction.

Thus, a concurrent process p_j completes infinitely many *add* operations. Before adding a new value v, p_j performs a *get* operation and attaches the returned view to the added value v. Moreover, eventually p_j will receive and include in V_j all values that appear in U. Thus, eventually, p_j will attach a view W such that $values(U) \subseteq W$ to every value it adds (line 8). As p_j is correct, $[v, W]$ will be included to V_i and p_i will return in line 23 – a contradiction. ◀

▶ **Theorem 6.** *Algorithm 1 implements a sequentially consistent add-only set in the anonymous non-instantaneous broadcast model.*

Proof. Fix an execution of the algorithm. Let H be the corresponding history.

By Lemma 2, all views returned by complete *get* operations in this execution can be totally ordered based on the growing containment relation. Let S be the corresponding sequential history of *get* operations. By the algorithm, all values that appear in these views are arguments of some (complete or incomplete) *add* operations. Thus, we can amend S by inserting each such operation $add(v)$ just before the first *get* in S that returns a view containing v. By Lemma 3 and 4, we can choose S to be consistent with local histories $H|i$ which end up with complete *get* operations.

If there are only finitely many complete *get* operations in H, all complete *add* operations in H whose arguments do not appear in views returned in H can be inserted after the last *get* operation in S without affecting legality.

Otherwise, if there are infinitely many *get* operations in H, we claim that the argument of every complete *add* operation in H appears in some view returned in H. Indeed, by the algorithm (line 9), every complete $add(v)$ operation by p_i broadcasts V_i that includes v and, eventually, every correct process will have v in its local view. Thus, there is a time after which every complete *get* operation returns a view containing v.

Thus, the resulting legal sequential history S is equivalent to a completion of H that contains all complete *add* and *get* operations and some incomplete *add* operations in H.

Finally, by Lemma 5, every operation invoked by a correct process returns. Thus, Algorithm 1 is a sequentially consistent implementation of add-only set. ◀

5 Colorless tasks in anonymous networks

Our sequentially consistent implementation of an add-only set allows us to prove several interesting computability results for *colorless tasks*.

Colorless tasks. A process invokes a distributed *task* with an *input* value and the task returns an *output* value, so that the inputs and the outputs across the processes respect the task specification. In a *colorless* task, processes are free to use each others' input and output values, so the task can be defined in terms of input sets and output sets.

Formally, a *colorless task* is defined through a set $\mathcal{I}$ of input sets, a set $\mathcal{O}$ of output sets, and a total relation $\Delta : \mathcal{I} \mapsto 2^{\mathcal{O}}$ that associates each input set with a set of possible output sets. We require that Δ is *carrier* map i.e., $\forall \tau, \sigma \in \mathcal{I} : \tau \subseteq \sigma \Rightarrow \Delta(\tau) \subseteq \Delta(\sigma)$. A colorless task is said to be *t-resiliently solvable* if there exists a t-resilient protocol that guarantees that, in every execution with t or fewer failures and input set $\sigma \in \mathcal{I}$, every correct process outputs a value such that the output set τ satisfies $\tau \in \Delta(\sigma)$. Check [18] for more details on the definition.

In this section, we show that the anonymous broadcast model is equivalent, from the colorless task computability viewpoint, to the non-anonymous read-write shared memory model.

Sequentially-consistent snapshot. We now describe an implementation of a *sequentially-consistent* snapshot object, that maintains a vector of ℓ values and exports two types of operations, `update`(i, v) and `snapshot`$()$, where $i = 0, 1, \ldots, \ell$, with the following sequential specification. Operation `update`(i, v) replaces the content of the i-th component of the object with value v, and `snapshot`$()$ returns the current state of the vector. Without loss of

Algorithm 2 implementation of sequentially-consistent snapshot: code for a process p_i

```
1  Shared variables:
2      SET: a sequentially-consistent add-only set

3  update(i,v):
4      S = SET.get()
5      t = maxTS(S)
6      SET.add([i,t+1,v])

7  snapshot():
8      S = SET.get()
9      v = recent(S)
10     return v

11 macro maxTS(S):
12     if S = ∅ then
13         return 0
14     return max{t | ∃i,v : [i,t,v] ∈ S}

15 macro recent(S):
16     for i in 1,...,ℓ do
17         snap[i] = recent_i(S)
18     return snap

19 macro recent_i(S):
20     if {(t,v) | ∃i : [i,t,v] ∈ S} = ∅ then
21         return ⊥
22     (t',v') = max{(t,v) | ∃i : [i,t,v] ∈ S}
23     return v'
```

generality, we assume that the values that can be stored in the vector are natural numbers. We also assume that each component of the object is initialized with a default value $\perp$.

Our implementation of a sequentially-consistent snapshot maintains a single shared sequentially consistent set `SET` and operates as follows. To update the i-th component by a value v, a process takes the current copy of the underlying add-only set, calculates the largest timestamp t, increments it, and then adds tuple $[i, t, v]$ to the set. To take a snapshot, the process first evaluates the current state S of the underlying add-only set. Then, the process calculates the most recent value of the each i-th component based on the lexicographic order on the set $\{(t, v) \mid [i, t, v] \in S\}$, where the empty set corresponds the initial value $\perp$. Overall, the implementation, presented in Algorithm 2, resembles an implementation of a linearizable register in the non-anonymous broadcast model [4].

▶ **Theorem 7.** *Algorithm 2 implements a sequentially-consistent snapshot in the anonymous non-instantaneous broadcast model.*

Proof. Fix an execution of the algorithm and let H be the corresponding history. In the executions, each `update`(i, v) is associated with a timestamp $\text{TS}(\texttt{update}(i, v)) = \texttt{maxTS}(S)+1$ for S in Line 4. For each `snapshot`(), underlyingSet(`snapshot`()) denotes the set S of Line 8. We write $(t, v) < (t', v')$ if and only if (t, v) is lexicographically smaller than (t', v').

We may assume, without loss of generality, that every process performs updates and snapshots alternatively, e.g., running a full-information protocol: the first update carries the input value of the process and every next update carries the outcome of the preceding snapshot operation.

We can now construct a sequential history S corresponding to H in the following manner: Let op_1 and op_2 be update or snapshot operations appeared in H.

- In the case of $op_1 = \mathtt{update}(i, v)$ and $op_2 = \mathtt{update}(i, v')$, op_1 precedes op_2 in S if and only if $(\mathrm{TS}(op_1), v) < (\mathrm{TS}(op_2), v')$.
- In the case of $op_1 = \mathtt{snapshot}()$ and $op_2 = \mathtt{snapshot}()$, op_1 precedes op_2 in S if and only if $\mathrm{underlyingSet}(op_1) \subseteq \mathrm{underlyingSet}(op_2)$.
- In the case of $op_1 = \mathtt{update}(i, v)$ and $op_2 = \mathtt{snapshot}()$, op_1 precedes op_2 in S if and only if the tuple $[i, t, v]$ that op_1 has added at Line 6 is in $\mathrm{underlyingSet}(op_2)$.

In all other cases, op_1 and op_2 can be arbitrarily ordered as long as the order keeps the process local histories of H. ◀

Since decision tasks do not concern the order of inputs or outputs, the following lemma holds:

▶ **Lemma 8.** *Assume that there is an anonymous shared-memory protocol that runs on top of an atomic snapshot object and t-resiliently solve a colorless task T. Then the very same protocol can be run on top of a sequentially-consistent snapshot object and still t-resiliently solve the colorless task T.*

Proof. Let P (resp. P') be the protocol that runs on top of the atomic snapshot object (resp. a sequentially-consistent object). Fix a set of inputs $\sigma \in \mathcal{I}$ and fix the execution E' of the protocol P' starting from σ. Let H' be the corresponding history of E' and S' be the sequential history that is equivalent to H'. Then, there exists an execution E of the protocol P that corresponds to the execution S'.

In executions E and E', every process p_i reads and writes the exactly same values in the same order. Thus, if the process terminates and makes output in E, the process must terminate and produce the same output in E'. By the above arguments, if the protocol P t-resiliently solves the given colorless task T, the protocol P' also t-resiliently solves the colorless task T. ◀

Therefore, every decision task that can be t-resiliently solved with an atomic snapshot object can be t-resiliently solved with a sequentially-consistent snapshot object.

Combining Theorem 6, Theorem 7, and Lemma 8 we obtain the following result.

▶ **Lemma 9.** *Every colorless task that is t-resiliently solvable in the anonymous shared-memory model is also t-resiliently solvable in the anonymous broadcast model, where $t < \frac{n}{2}$.*

We can prove the converse of Lemma 9 by [13, Theorem 3].

▶ **Theorem 10** ([13, Theorem 3])**.** *A colorless task is t-resiliently solvable in the anonymous shared-memory model if and only if it is t-resiliently solvable in the non-anonymous one.*

▶ **Lemma 11.** *Every colorless task that is t-resiliently solvable in the anonymous broadcast model is also t-resiliently solvable in the anonymous shared-memory model, where $t < \frac{n}{2}$.*

Proof. If a colorless task T is t-resiliently solvable in the anonymous broadcast model, it is obviously t-resiliently solvable in the non-anonymous broadcast model. Then, T is also t-resiliently solvable in the non-anonymous shared-memory model because the non-anonymous shared-memory model simulates the non-anonymous broadcast model, where $t < \frac{n}{2}$ [4]. Thus, by Theorem 10, the colorless task T is t-resiliently solvable in the anonymous shared-memory model. ◀

Let us note that the above argument is necessary because, in the anonymous setting, the shared-memory model does not simulate the broadcast model. In the anonymous shared-memory model, processes cannot know the number of clone processes if they take steps alternately and keep in the identical state. On the other hand, in the anonymous broadcast model, each process eventually receives the messages sent by clone processes by an amount equal to the number of the clones.

▶ **Theorem 12.** *A colorless task T is t-resiliently solvable in the anonymous broadcast model if and only if it is t-resiliently solvable in the anonymous shared-memory model, where $t < \frac{n}{2}$.*

The theorem says that the anonymous broadcast model is equivalent to the anonymous shared-memory model from the viewpoint of colorless task solvability when less than majority of processes may fail.

6 Concluding remarks: on linearizability in anonymous networks

We showed that the anonymous broadcast model does not allow *linearizable* implementations of nontrivial object types (Theorem 1). On the positive side, we suggested to consider sequential consistency and proposed a sequentially consistency implementation of *add-only set*. While sequential consistency is good enough for exploring *one-shot* task computability (Section 5), it may not be an attractive property for *long-lived* abstractions that can be *composed* in a larger context. Unlike linearizability, the property does not preserve *real-time precedence* of operations: precisely because of this, sequentially consistent implementations do not compose [17].

A closer look at the proof of Theorem 1 reveals that sequential consistency might be a very rough fix to obviate the impossibility. Indeed, in the constructed non-linearizable history, a clone of a process that completed its operation earlier cannot distinguish the current state from the initial one and, thus, must return a "stale", inconsistent response. However, *modulo* operations executed by the clone, the constructed history can be seen as a linearizable one that preserves real-time ordering of "original" (non-cloning) operations.

Intuitively, we can think of a straightforward modification of our *add-only set* implementation that seems to enable this kind of relaxed linearizability. Before returning, an *add* operation may ensure that the added value is accepted by a majority of processes.

An interesting open question is whether this intuition is justified: can we come up with a *composable* relaxation of linearizability that applies to the anonymous broadcast model?

References

1. Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic Snapshots of Shared Memory. *JACM*, 40(4):873–890, 1993.
2. Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006.
3. James Aspnes, Faith Ellen Fich, and Eric Ruppert. Relationships between broadcast and shared memory in reliable anonymous distributed systems. *Distributed Computing*, 18(3):209–219, 2006.
4. Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing Memory Robustly in Message-passing Systems. *J. ACM*, 42(1):124–142, January 1995. doi:10.1145/200836.200869.
5. François Bonnet and Michel Raynal. The Price of Anonymity: Optimal Consensus Despite Asynchrony, Crash, and Anonymity. *ACM Trans. Auton. Adapt. Syst.*, 6(4):23:1–23:28, October 2011. doi:10.1145/2019591.2019592.

6 Zohir Bouzid, Michel Raynal, and Pierre Sutra. Anonymous obstruction-free (n, k)-set agreement with n-k+1 atomic read/write registers. *Distributed Computing*, 31(2):99–117, 2018.

7 Zohir Bouzid, Pierre Sutra, and Corentin Travers. Anonymous Agreement: The Janus Algorithm. In *Principles of Distributed Systems - 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings*, pages 175–190, 2011.

8 Claire Capdevielle, Colette Johnen, Petr Kuznetsov, and Alessia Milani. On the uncontended complexity of anonymous agreement. *Distributed Computing*, 30(6):459–468, 2017.

9 S. Chaudhuri. More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation*, 105(1):132–158, 1993.

10 Tom Chothia and Konstantinos Chatzikokolakis. A Survey of Anonymous Peer-to-Peer File-Sharing. In *Proceedings of Satellite Workshop of the International Conference on Embedded and Ubiquitous Systems (EUS)*, pages 744–755, 2005.

11 Carole Delporte-Gallet and Hugues Fauconnier. Two Consensus Algorithms with Atomic Registers and Failure Detector Ω. In *Proceedings of the 10th International Conference on Distributed Computing and Networking (ICDCN)*, volume 5408 of *Lecture Notes in Computer Science (LNCS)*, pages 251–262, 2009.

12 Carole Delporte-Gallet, Hugues Fauconnier, Petr Kuznetsov, and Eric Ruppert. On the Space Complexity of Set Agreement? In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 271–280, 2015.

13 Carole Delporte-Gallet, Hugues Fauconnier, Sergio Rajsbaum, and Nayuta Yanagisawa. A Characterization of t-Resilient Solvable Colorless Tasks in Anonymous Shared-Memory Model. In *SIROCCO2018 (to appear)*, 2018. Technical report: `https://arxiv.org/abs/1712.04393`.

14 Carole Delporte-Gallet, Hugues Fauconnier, Sergio Rajsbaum, and Nayuta Yanagisawa. An anonymous wait-free weak-set object implementation. In *NETYS2018 (to appear)*, 2018.

15 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, April 1985.

16 Rachid Guerraoui and Eric Ruppert. Anonymous and fault-tolerant shared-memory computing. *Distributed Computing*, 20(3):165–177, 2007.

17 Maurice Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):123–149, 1991.

18 Maurice Herlihy, Dmitry Kozlov, and Sergio Rajsbaum. *Distributed computing through combinatorial topology*. Morgan Kaufmann, 2013.

19 Maurice Herlihy and Sergio Rajsbaum. A classification of wait-free loop agreement tasks. *Theoretical Computer Science*, 291(1):55–77, 2003.

20 Nayuta Yanagisawa. Wait-Free Solvability of Colorless Tasks in Anonymous Shared-Memory Model. *Theory of Computing Systems*, pages pp 1–18, November 2017. `doi:10.1007/s00224-017-9819-0`.