Distrib. Comput. (2007) 20:165–177
DOI 10.1007/s00446-007-0042-0

# Anonymous and fault-tolerant shared-memory computing

**Rachid Guerraoui · Eric Ruppert**

Received: 17 November 2005 / Accepted: 23 October 2006 / Published online: 4 September 2007
© Springer-Verlag 2007

**Abstract** The vast majority of papers on distributed computing assume that processes are assigned unique identifiers before computation begins. But is this assumption necessary? What if processes do not have unique identifiers or do not wish to divulge them for reasons of privacy? We consider asynchronous shared-memory systems that are anonymous. The shared memory contains only the most common type of shared objects, read/write registers. We investigate, for the first time, what can be implemented deterministically in this model when processes can fail. We give anonymous algorithms for some fundamental problems: time-stamping, snapshots and consensus. Our solutions to the first two are wait-free and the third is obstruction-free. We also show that a shared object has an obstruction-free implementation if and only if it satisfies a simple property called idempotence. To prove the sufficiency of this condition, we give a universal construction that implements any idempotent object.

**Keywords** Anonymous · Shared memory · Timestamps · Snapshots · Consensus

R. Guerraoui (✉)
Distributed Programming Laboratory, School of Computer and Communication Sciences, Ecole Polytechnique, Fédérale de Lausanne, CH 1015, Lausanne, Switzerland
e-mail: rachid.guerraoui@epfl.ch

E. Ruppert
Department of Computer Science and Engineering, York University, 4700 Keele Street Toronto, Toronto, ON, Canada M3J 1P3

## 1 Introduction

Distributed computing typically studies what can be computed by a system of $n$ processes that can fail independently. Variations on the level of synchrony, the means of inter-process communication, failure modes of the system, and other parameters have led to an abundant literature. In particular, a prolific research trend has explored the capabilities of a system of crash-prone asynchronous processes communicating through basic read/write objects (registers).

To fully understand theoretical models of distributed computing, it is important to precisely measure the impact of their underlying assumptions. Virtually all of the literature on distributed computing assumes that processes have distinct identities. Thus, it is interesting to investigate what can be done without the assumption of distinct identities. A system is called *anonymous* if processes are programmed identically [5,7,8,13,26,31]. In particular, processes do not have identifiers.

Besides intellectual curiosity, it is also appealing to revisit this fundamental assumption for practical reasons. Indeed, certain systems, like sensor networks, consist of mass-produced tiny agents that might not even have identifiers [4]. Others, like web servers [32] and peer-to-peer file sharing systems [12], sometimes mandate preserving the anonymity of the users and forbid the use of any form of identity for the sake of privacy. (See [11] for a discussion of anonymous computing used for privacy.)

Consider a server that houses a collection of shared-memory objects. Users wish to access these objects to carry out a shared-memory distributed algorithm. However, the users may not wish to divulge their identities to the server or to one another, or even allow the server to observe that two different operations on the shared objects are coming from the same source. Instead of revealing their identities to the server, users

might enlist the help of a trusted third party that provides an anonymous proxy service. This party forwards all processes' invocations to the server (stripped of the processes' identifiers) and then forwards the server's responses back to the processes. The trusted third party can be approximated by a decentralized mechanism called onion routing [19]. Whereas this scheme makes anonymous shared-memory computing possible, it is not clear what can actually be done in an anonymous system.

There has been work on anonymous message-passing systems, starting with Angluin [3]. The small amount of research that has looked at anonymous shared-memory systems assumed failure-free systems or the existence of a random oracle to build randomized algorithms. (See Sect. 2 for a discussion of this work.)

We explore in this paper the types of shared objects that can be implemented *deterministically* in an anonymous, asynchronous shared-memory system. We assume that any number of unpredictable crash failures may occur. The shared memory is composed of registers that are (multi-reader and) multi-writer, so that every process is permitted to write to every register. In contrast, usage of single-writer registers would violate total anonymity by giving processes at least some rudimentary sense of identity: processes would know that values written into the same register at different times were produced by the same process.

Several properties have been defined to describe the progress made by an algorithm regardless of process crashes or asynchrony. The strongest is *wait-freedom* [21], which requires *every* non-faulty process to complete its algorithm in a finite number of its own steps. However, wait-free algorithms are often either provably impossible or too inefficient to be practical. In many settings, a weaker progress guarantee is sufficient. The *non-blocking* property (sometimes called lock-freedom) is one such guarantee, ensuring that, eventually, *some* process will complete its algorithm. It is weaker than wait-freedom because it permits individual processes to starve. A third condition that is weaker still is *obstruction-freedom* [22], which can be very useful when low contention is expected to be the common case, or if contention-management is used. Obstruction-freedom guarantees that a process will complete its algorithm whenever it has an opportunity to take enough steps without interruption by other processes. If only registers are available for communication and algorithms are deterministic, obstruction-freedom is strictly weaker than the non-blocking property. However, a randomized contention manager can be used to turn an obstruction-free algorithm into a wait-free one.

Some problems, such as leader election, are clearly impossible to solve anonymously because symmetry cannot be broken; if processes run in lockstep, they will perform exactly the same sequence of operations. However, we show that some interesting problems *can* be solved without breaking symmetry.

We first consider *timestamps*, which are frequently used to help processes agree on the order of various events. Objects such as fetch&increment and counters, which have been used traditionally for creating timestamps, have no obstruction-free implementation in our anonymous model (as we shall see in Theorem 12). We introduce a weaker object called a *weak counter*, which provides sufficiently good timestamps for our applications. We construct, in Sect. 4, an efficient, wait-free implementation of a weak counter.

In non-anonymous systems, the *snapshot* object [1,2,6] is probably the most important example of a shared object that has a wait-free implementation from registers. It is an abstraction of the problem of obtaining a consistent view of many registers while they are being concurrently updated by other processes. There are many known implementations of snapshot objects but, to our knowledge, all do make essential use of process identities. Wait-free algorithms generally rely on helping mechanisms, in which fast processes help the slow ones complete their operations. One of the challenges of anonymity is the difficulty of helping other processes when it is not easy to determine who needs help. In Sect. 5, we show that a wait-free snapshot implementation does exist and has fairly efficient time complexity. The timestamps provided by the weak counter are essential in this construction. We also give a non-blocking implementation with better space complexity.

In non-anonymous systems, most objects have no wait-free (or even non-blocking) implementation [21]. However, it is possible to build an obstruction-free implementation of any object by using a subroutine for *consensus*, which is a cornerstone of distributed computing that does itself have an obstruction-free implementation [22]. Consensus also arises in a wide variety of process-coordination tasks. There is no (deterministic) wait-free implementation of consensus using registers, even if processes do have identifiers [21,29]. In Sect. 6, we note that an obstruction-free anonymous consensus algorithm can be obtained by simply derandomizing the randomized algorithm of Chandra [14]. The resulting algorithm uses unbounded space. We then give a new algorithm that uses a bounded number of registers, with the help of our snapshots.

Finally, we give a complete characterization of the types of objects that have obstruction-free implementations in our model in Sect. 7. An object can be implemented if and only if it is idempotent, i.e., applying any permitted operation twice in a row (with the same arguments) has the same effect as applying it once. This means that both consecutive invocations return the same response, and the state of the object after the two invocations is indistinguishable from the state it has after just one invocation. (A formal definition of idempotence appears in Sect. 7.) Examples of idempotent objects

**Table 1** Summary of implementations, where $n$ is the number of processes, $k$ is the number of operations invoked, and $d$ is the number of possible inputs to consensus

| Theorem | Implemented object | Using | Number of registers | Progress | Uses |
|---|---|---|---|---|---|
| 3 | Weak counter | Binary registers | $O(k)$ | Non-blocking | |
| 1 | Weak counter | Registers | $O(k)$ | Wait-free | |
| [16] | Weak counter | Registers | $n$ | Wait-free | |
| [16] | Weak counter | Registers | $O(n^2)$ | Bounded wait-free | |
| 4 | $m$-Component snapshot | Registers | $m$ | Non-blocking | |
| 5 | $m$-Component snapshot | Registers | $O(m+k)$ | Wait-free | 1 |
| 5 | $m$-Component snapshot | Registers | $m+n$ | Wait-free | [16] |
| 5 | $m$-Component snapshot | Registers | $O(m+n^2)$ | Bounded wait-free | [16] |
| 6 | Binary consensus | Binary registers | Unbounded | Obstruction-free | |
| 7 | Binary consensus | Registers | $O(n)$ | Obstruction-free | 4 |
| 9 | Consensus | Binary registers | Unbounded | Obstruction-free | 6, 8 |
| 9 | Consensus | Registers | $O(n \log d)$ | Obstruction-free | 7, 8 |
| 12 | Idempotent object | Binary registers | Unbounded | Obstruction-free | 3, 9 |
| 14 | Idempotent object | Registers | Object-dependent | Obstruction-free | 3, 7 |

include registers, snapshot objects, sticky bits and resettable consensus objects. We use a symmetry argument to show the idempotence condition is necessary for the existence of an anonymous implementation. To prove sufficiency, we give a universal construction that implements any idempotent object, using our weak counter object and our consensus algorithm.

To summarize, we show that the anonymous asynchronous shared-memory model has some, perhaps surprising, similarities to the non-anonymous model, but there are also some important differences. We construct a wait-free algorithm for snapshots and an obstruction-free algorithm for consensus that uses bounded space. Not every type of object has an obstruction-free anonymous implementation, however. We give a characterization of the types that do. Table 1 lists all anonymous implementations given in this paper, indicating which implementations are used as subroutines for others. Ellen et al. [16] gave some anonymous timestamp algorithms with bounded space complexity, and those can be combined with the wait-free snapshot algorithm given here to improve the snapshot's complexity. Those improvements are also listed in the table and are discussed further in Sect. 8.

## 2 Related work

Some research has studied anonymous shared-memory systems under the assumption that no failures can occur. Attiya et al. [8] gave a characterization of the tasks that are solvable without failures using registers if $n$ is not known. The characterization is the same if $n$ is known [15]. Consensus is solvable in these models, but it is not solvable if the registers cannot be initialized by the programmer [25]. Aspnes et al. [5] looked at failure-free models with other types of objects, such as counters. They also characterized which shared-memory models can be implemented if communication is through anonymous broadcasts, showing the broadcast model is equivalent to having shared counters and strictly stronger than shared registers. Johnson and Schneider [26] gave leader election algorithms using versions of single-writer snapshots and test&set objects.

There has also been some research on randomized algorithms for anonymous shared-memory systems with no failures. The naming problem, where processes must choose unique names for themselves, is a key problem since it essentially turns an anonymous system into a non-anonymous one. Processes can randomly choose names, which will be unique with high probability. Registers can be used to detect when the names chosen are indeed unique, thus guaranteeing correctness whenever the algorithm terminates, which happens with high probability [28,34]. Two papers gave randomized renaming algorithms that have finite expected running time, and hence terminate with probability one [17,27].

Randomized algorithms for systems with crash failures have also been studied. Panconesi et al. [31] gave a randomized wait-free algorithm that solves the naming problem using single-writer registers, which give the system some ability to distinguish between different processes' actions. Several impossibility results have been shown for randomized naming using only multi-writer registers [13,17,27]. Interestingly, Buhrman et al. [13] gave a randomized wait-free anonymous algorithm for consensus in this model that is based on Chandra's randomized consensus algorithm [14]. Thus, producing unique identifiers is strictly harder than

consensus in the randomized setting. Aspnes et al. [7] extended the consensus algorithm of Buhrman et al. to a setting with infinitely many processes.

Solving a decision task can be viewed as a special case of implementing objects: each process accesses the object, providing its input as an argument, and later the object responds with the output the process should choose. Herlihy and Shavit [23] described how their characterization of decision tasks that have wait-free solutions in non-anonymous systems can be extended to systems with a kind of anonymity: processes have identifiers but are only allowed to use them in very limited ways. Their characterization uses tools from algebraic topology. Herlihy gave a *universal construction* which describes how to create a wait-free implementation of any object type using consensus objects [21]. Processes use consensus to agree on the exact order in which the operations are applied to the implemented object. Although this construction requires identifiers, it was the inspiration for our obstruction-free construction in Sect. 7. Recently, Bazzi and Ding [10] introduced, in the context of Byzantine systems, non-skipping timestamps, a stronger abstraction than what we call a weak counter. (The specification of our weak counter does not preclude skipping values.)

Obstruction-free shared memory algorithms have been studied by Attiya et al. [9] assuming unique process identities. In particular, their consensus algorithm can be viewed as a race between processes and it relies on the fact that every process writes in its own register. The consensus algorithm we describe in this paper can rather be viewed as a race between values, and all processes may write in all registers. They also show that *every* object type has an obstruction-free implementation from registers in the non-anonymous model, which contrasts with our results in Sect. 7 for the anonymous case.

## 3 Model

We consider an *anonymous* system, where a collection of $n$ processes execute identical algorithms. In particular, the processes do not have identifiers. The system is *asynchronous*, which means that processes run at arbitrarily varying speeds. It is useful to think of processes being allocated steps by an adversarial scheduler. Algorithms must work correctly in all possible schedules. Processes are subject to *crash failures*: they may stop taking steps without any warning. The algorithms we consider are *deterministic*.

Processes communicate with one another by accessing shared data structures, called *objects*. The *type* of an object specifies what states it can have and what operations may be performed on it. The programmer chooses the initial state of the objects used. Except for our weak counter object in Sect. 4, all objects are linearizable (atomic) [24]: although operations on an object take some interval of time to complete, each appears to happen at some instant between its invocation and response. An operation atomically changes the state of an object and returns a response to the invoking process. (The weak counter object can be viewed as a set-linearizable object [30].) We consider *oblivious* objects: all processes are permitted to perform the same set of operations on it and its response to an operation does not depend on the identity of the invoking process. (Non-oblivious objects would be somewhat inconsistent with the notion of totally anonymous systems, since processes would have to identify themselves when they invoke an operation.)

Some types of objects are provided by the system and all other types needed by a programmer must be implemented from them. An *implementation* specifies the code that must be executed to perform each operation on the implemented object. Since we are considering anonymous systems, all processes execute identical code to perform a particular operation. (We refer to such an implementation as an anonymous implementation.) The implementation must also specify how to initialize the base objects to represent any possible starting state of the implemented object. Implemented objects should appear, from the user's point of view, as if they are provided as base objects by the system. In particular, they should be linearizable too. A *configuration* is a description of the system at some instant in time. It is comprised of the state of each shared object used in the implementation and the local state of every process.

We assume the shared memory contains the most basic kind of objects: *registers*, which provide two types of operations. A *read* operation returns the state of the object without changing it. A *write*($v$) changes the state to $v$ and returns *ack*. Every process can access every register. If the set of possible values that can be stored is finite, the register is *bounded*; otherwise it is *unbounded*. A *binary* register has only two possible states. When describing our algorithms in pseudocode, names of shared objects begin with upper-case letters, and names of the process's private variables begin with lower-case letters.

## 4 Weak counters

A *weak counter* provides a single operation, GetTimestamp, which returns an integer. It has the property that if one operation precedes another, the value returned by the later operation must be larger than the value returned by the earlier one. (Two concurrent GetTimestamp operations may return the same value.) This object will be used as a building block for our implementation of snapshots in Sect. 5 and our characterization of implementable types in Sect. 7. It is used in those algorithms to provide timestamps to different operations.

The weak counter is essentially a weakened form of a fetch&increment object: a fetch&increment object has the additional requirements that the values returned should be distinct and consecutive. It is known that a fetch&increment object has no wait-free implementation from registers, even if processes have identifiers [21]. By considering our weaker version, we have an object that is implementable, and still strong enough for our purposes. Our weak counter implementations also generate timestamps that do not grow too quickly: the value returned to any operation does not exceed the number of invocations that have occurred so far. Although this property is not necessary to satisfy the specification of the object, it is useful in bounding the space used to store timestamps.

We give an anonymous, wait-free implementation of a weak counter from unbounded registers. A similar but simpler construction, which provides an implementation that satisfies the weaker non-blocking progress property, but uses only binary registers, is then described briefly. Processes must know $n$, the number of processes in the system, (or at least an upper bound on $n$) for the wait-free implementation, but this knowledge is not needed for the non-blocking case.

Our wait-free implementation uses an infinite array $A[1, 2, \ldots]$ of binary registers, each initialized to $\perp$. (The array can be finite if the number of operations to be applied on the counter is known in advance.) To obtain a counter value, a process locates the first entry of the array that is $\perp$, changes it to $\top$, and returns the index of this entry (see Fig. 1). The key property for correctness is the following invariant: if $A[k] = \top$, then all entries in $A[1 \ldots k]$ are $\top$. To locate the first $\perp$ in $A$ efficiently, the algorithm uses a binary search. Starting from the location $a$ returned by the process's previous GETTIMESTAMP operation, the algorithm probes locations $a + 1, a + 3, a + 7, \ldots, a + 2^i - 1, \ldots$ until it finds a $\perp$ in some location $b$. (For the first operation by the process, we initialize $a$ to 1.) We call this portion of the algorithm, corresponding to the first loop in the pseudocode, phase 1. The process then executes a binary search of $A[a \ldots b]$ in the second loop, which constitutes phase 2.

To ensure processes cannot enter an infinite loop in phase 1 (while other processes write more and more $\top$'s into the array), we incorporate a helping mechanism. Whenever a process writes a $\top$ into an entry of $A$, it also writes the index of the entry into a shared register $L$ (initialized to 0). A process may terminate early if it sees that $n$ writes to $L$ have occurred since its invocation. In this case, it returns the largest value it has seen in $L$. The local variables $j$ and $t$ keep track of the number of times the process has seen $L$ change, and the largest value the process has seen in $L$, respectively.

**Theorem 1** *Figure 1 gives a wait-free, anonymous implementation of a weak counter from registers.*

```
GETTIMESTAMP
 1  b ← a + 1
 2  ℓ ← L
 3  t ← ℓ
 4  j ← 0
 5  loop until A[b] = ⊥
 6          if L ≠ ℓ
 7             then ℓ ← L
 8                  t ← max(t, ℓ)
 9                  j ← j + 1
10                  if j ≥ n
11                     then a ← b + 1
12                          return t and halt
13                  end if
14          end if
15          b ← 2b − a + 1
16  end loop
17  loop until a = b
18          mid ← (a+b−1)/2        ▷ This is always an integer
19          if A[mid] = ⊥ then b ← mid
20          else a ← mid +1
21          end if
22  end loop
23  write ⊤ to A[b]
24  L ← b
25  return b
```

**Fig. 1** Wait-free implementation of a weak counter from registers

*Proof* We first give three simple invariants.

*Invariant 1*: *For each process's value of a, if* $a > 1$, *then* $A[a - 1] = \top$.

Once $\top$ is written into an entry of $A$, that entry's value will never change again. It follows that line 20 maintains Invariant 1. Line 11 does too, since the preceding iteration of line 5 found that $A[b] = \top$.

*Invariant 2*: *If* $A[k] = \top$, *then* $A[k'] = \top$ *for all* $k' \leq k$.

Entries never change from $\top$ to $\perp$, so Invariant 2 follows from Invariant 1: whenever line 23 is executed, we have $a = b$, so $A[b - 1]$ is already $\top$, by Invariant 1.

*Invariant 3*: *Whenever a process P executes line* 17 *during a* GETTIMESTAMP *operation op, P's value of b has the property that* $A[b]$ *was equal to* $\perp$ *at some earlier time during op.*

This is true the first time line 17 is executed, by the exit condition of the first loop. Every time $b$ is changed in line 19, it is changed to the index of a location which has just been seen to contain $\perp$.

*Wait-freedom*: To derive a contradiction, assume there is an execution where some operation by a process $P$ runs forever without terminating. This can only happen if there is an infinite loop in Phase 1, so an infinite number of $\top$'s are written into $A$ during this execution. This means that an infinite number of writes to $L$ will occur. Suppose some process $Q$ writes a value $x$ into $L$. Before doing so, it must write $\top$ into $A[x]$. Thus, any subsequent invocation of GETTIMESTAMP by $Q$ will never see $A[x] = \perp$. It follows from Invariant 3 that $Q$ can never again write $x$ into $L$. Thus, $P$'s operation will eventually see $n$ different values in $L$ and terminate, contrary to the assumption.

*Correctness*: Finally, we prove that the implementation satisfies the specification of a weak counter. Suppose one GETTIMESTAMP operation $op_1$ completes before another one, $op_2$, begins. Let $r_1$ and $r_2$ be the values returned by $op_1$ and $op_2$, respectively. We must show that $r_2 > r_1$. If $op_1$ terminates in line 12, then, at some earlier time, some process wrote $r_1$ into $L$ and also wrote $\top$ into $A[r_1]$. If $op_1$ terminates in line 25, it is also clear that $A[r_1] = \top$ when $op_1$ terminates.

If $op_2$ terminates in line 25, then $A[r_2]$ was $\perp$ at some time during $op_2$, by Invariant 3. Thus, by Invariant 2, $r_2 > r_1$. If $op_2$ terminates in line 12, it returns a value that some process wrote into $L$. In this case, $op_2$ has seen the value in $L$ change $n$ times during its run, so at least two of the changes were made by the same process. Since a GETTIMESTAMP operation can write to $L$ at most once, at least one of those changes was made by an operation $op_3$ that started after $op_2$ began (and hence after $op_1$ terminated). Since $op_3$ terminated in line 25, we have already proved that the value $r_3$ that $op_3$ returns (and writes into $L$) must be greater than $r_1$. But $op_2$ returns the largest value it sees in $L$, so $r_2 \geq r_3 > r_1$. Thus $op_2$ returns a larger result than $op_1$ in either case, as required. □

In any finite execution in which $k$ GETTIMESTAMP operations are invoked, at most $O(k)$ of the registers are ever accessed, and the worst-case time for any operation is $O(\log k)$. This is because each operation writes to at most one location in $A$, so the first loop must terminate when $b$ becomes bigger than $k$. If the implementation runs forever, $k$ can grow arbitrarily large, so this implementation is wait-free, but not bounded wait-free. We use an amortized analysis to prove the stronger bound of $O(\log n)$ on the *average* time per operation in any finite execution. Intuitively, if some process $P$ must perform a phase 1 that is excessively long, we can charge its cost to the many operations that must have written into $A$ since $P$ did its previous operation.

**Proposition 2** *If $n$ processes perform a total of $k$ invocations of the* GETTIMESTAMP *algorithm in Fig. 1, the total number of steps by all processes is $O(k \log n)$ and $O(k)$ registers are accessed.*

*Proof* To do this amortized analysis, we shall count only the number of locations probed in line 5, since the time to perform the entire operation is proportional to this number.

Each time a process writes to an entry $A[i]$, it stores 4 credits in that location. Consider an operation $op$ that performs $k$ probes during phase 1, testing locations $a+1, a+3, a+7, \ldots, a+2^k-1$. The first $\log n$ probes are billed to $op$ itself. For $\log n < i \leq k$, the cost of the $i$th probe is paid for using $2^{-(i-2)}$ units of credit from each of the $2^{i-2}$ locations $a+2^{i-2}, \ldots, a+2^{i-1}-1$ (which must all contain $\top$, and therefore have credit stored on them). Since $i > \log n$, the amount billed to each location for the probe is $2^{-i+2} < 4 \cdot 2^{-\log n} = \frac{4}{n}$.

It remains to check that no location is charged more than 4 units of credit throughout the execution. Clearly, no two probes during the same operation are billed to the same location. A process $P$ probes location $a+2^i-1$ during phase 1 of an operation $op$ only if location $q = a+2^{i-1}-1$ already contains $\top$. This means that, either in line 11 or during phase 2 of $op$, $a$ will be changed to a value larger than $q$. If $P$ later performs another operation $op'$, it will not bill any of the probes done during $op'$ to any location smaller than $a$. It follows that process $P$ never bills two probes to the same location, so the amount billed to any location is at most $n \cdot \frac{4}{n} = 4$. Therefore, the amortized number of probes per operation is at most $4 + \log n \in O(\log n)$.

If there are $k$ invocations, at most $k$ registers will be written with non-$\perp$ values, so no process will ever access a register whose index is beyond $2k$. □

To see that the time analysis in the preceding proof is tight (when $k > n$), consider an execution where the following sequence is repeated $\lfloor \frac{k}{n} \rfloor$ times: process $P_1$ does $n-1$ complete GETTIMESTAMP operations, and then processes $P_2$ to $P_n$ each perform one GETTIMESTAMP operation in lock step. This execution has a total of $\Omega(\frac{k}{n}(n+(n-1)\log n)) = \Omega(k \log n)$ steps.

If we do not require the weak counter implementation to be wait-free, we do not need the helping mechanism. Thus, we can omit lines 2–4, 6–14 and 24, which allow a process to terminate early if it ever sees that $n$ changes to the shared register $L$ occur. This yields a non-blocking implementation that uses only *binary* registers. The proof of correctness is a simplified version of the proof of Theorem 1, and the analysis is identical to the proof of Proposition 2.

**Theorem 3** *There is a non-blocking, anonymous implementation of a weak counter from binary registers. In any execution with $k$ invocations of* GETTIMESTAMP *in a system of $n$ processes, the total number of steps is $O(k \log n)$ and $O(k)$ registers are accessed.*

## 5 Snapshot objects

The snapshot object [1,2,6] is an extremely useful abstraction of the problem of getting a consistent view of several registers when they can be concurrently updated by other processes. It has wait-free (non-anonymous) implementations from registers, and has been widely used as a basic building block for other algorithms. A snapshot object consists of a collection of $m > 1$ components and supports two kinds of operations: a process can update the value stored in a component and atomically scan the object to obtain the values of all the components. Since we are interested in anonymous systems, we consider the multi-writer version, where any process can update any component. Snapshot objects have been

```
UPDATE(i, x)
1  t ← GETTIMESTAMP
2  v ← SCAN
3  write (x, v, t) in R_i

SCAN
1  t ← GETTIMESTAMP
2  loop
3          read R_1, R_2, ..., R_m
4          if some register contained (*, v, t') with t' ≥ t
5             then return v
6          elseif n sets of reads gave the same results
7             then return the first field of each value in such a set
8          end if
9  end loop
```

**Fig. 2** Wait-free implementation of a snapshot object from registers

extensively studied, and many algorithms exist to implement snapshots, but all use process identifiers. However, a simple modification of the classic non-blocking snapshot algorithm for non-anonymous systems [1] yields an anonymous non-blocking algorithm.

One register is used to represent each component of the snapshot object. Each process keeps a local variable $t$ that stores a timestamp. To perform an UPDATE on a component with value $v$, a process simply writes $(t, v)$ into the corresponding register and increments its value of $t$. To SCAN, a process repeatedly reads all of the registers until it sees exactly the same set of values $(t_1, v_1), (t_2, v_2), \ldots, (t_m, v_m)$ in all of the registers during $q$ sets of reads, where $q = m(n-1)+2$. When this happens, the SCAN returns $(v_1, v_2, \ldots, v_m)$.

**Proposition 4** *The algorithm described in the preceding paragraph is a non-blocking, anonymous implementation of an $m$-component snapshot object from $m$ registers.*

*Proof* We show that the algorithm makes progress and is correctly linearizable.
*Non-blocking property*: UPDATES terminate in a single step. A SCAN can only be prevented from terminating if UPDATES are constantly being completed.
*Linearizability*: Consider a SCAN operation that terminates. We describe how to linearize the SCAN. For $1 \leq i < q$, let $T_i$ be the moment just after the $i$th identical set of reads is completed. Since a value-timestamp pair can be written only once by a process, that pair can be written into a register at most $n-1$ times during the SCAN. So, for any $j$, the value of the $j$th register is different from $(t_j, v_j)$ at most $n-1$ of the times $T_1, \ldots, T_{q-1}$. Since $q = m(n-1)+2$, there is one time $T_i$ at which, for all $j$, the $j$th register contains $(t_j, v_j)$. Linearize the SCAN at that moment. □

More surprisingly, we show that an algorithm for (non-anonymous) wait-free snapshots [1] can also be modified to work in an anonymous system (see Fig. 2). The original algorithm could create a unique timestamp for each UPDATE operation. We use our weak counter to generate timestamps that are not necessarily distinct, but are sufficient for implementing the snapshot object. The non-uniqueness of the identifiers imposes a need for more iterations of the loop than in the non-anonymous algorithm. Our algorithm uses $m$ (large) registers, $R_1, \ldots, R_m$, and one weak counter, which can be implemented from registers, by Theorem 1. Each register $R_i$ will contain a value of the component, a view of the entire snapshot object and a timestamp.

**Theorem 5** *The algorithm in Fig. 2 is an anonymous, wait-free implementation of a snapshot object from registers. The average number of steps per operation in any finite execution is $O(mn^2)$.*

*Proof* We prove that the algorithm is wait-free and linearizable.
*Wait-freedom*: We need only prove that the loop in the SCAN routine eventually terminates. Consider the time $T$ when the SCAN finishes executing line 1 and receives its timestamp $t$ from the weak counter. The only UPDATES that can write a timestamp less than or equal to $t$ after time $T$ are those UPDATES that had already started before $T$, since any UPDATE that begins after $T$ will receive a larger timestamp. Thus, at most $n-1$ changes to the registers $R_1, \ldots, R_m$ can be observed by the SCAN before the first termination condition is satisfied. If the first condition is never satisfied, then after at most $n(n-1)+1$ iterations of the loop, the SCAN will see the same values in $n$ sets of reads, so the loop can terminate using the second termination condition. Thus, each operation must terminate within $O(mn^2)$ steps, plus the time required for the GETTIMESTAMP operation. It follows from Proposition 2 that the total number of steps in an execution with $k$ invocations of UPDATES and SCANS is $O(k \log n + kmn^2) = O(kmn^2)$.
*Linearizability*: To prove the implementation is correct, we describe how to linearize all of the snapshot operations, including the SCANS embedded in UPDATE operations. An UPDATE operation is linearized at the moment it writes in line 3.

Consider a process $P$ that performs a SCAN that receives timestamp $t$ in line 1. Suppose that $P$ sees identical results $r_1, \ldots, r_m$ in $n$ sets of reads. Suppose the first of the $n$ sets of reads begins at time $T$. None of the triples $r_i$ will contain a timestamp greater than $t$; otherwise $P$ would have terminated after the first set. So, any UPDATE that writes the triple $r_i$ into register $R_i$ after $T$ must already have been pending at time $T$. There are at most $n-1$ such UPDATES. This means that during one of the identical sets of reads, no UPDATE

performed a write, so throughout the interval of time $P$ takes to do that set of reads, the value of $R_i$ was $r_i$, for all $i$. If we linearize the SCAN at some time during this interval, it will return a valid result.

It remains to linearize a SCAN $S$ that exits the loop using the first termination condition. Let $t$ be the timestamp obtained by $S$, and $t' \geq t$ be the timestamp associated with the view $v$ returned by $S$. Assume that we have already chosen correct linearization points for all SCANS that terminated before $S$ terminates. In particular, we have chosen a linearization point for the SCAN $S'$ that was embedded in an UPDATE operation $U$ that wrote the value $(*, v, t')$ that was read by $S$. We linearize $S$ immediately after $S'$. Since the two SCANS return the same result, this choice satisfies the correctness property of the snapshot object. This linearization point is clearly before $S$ returns. We must check that it is also after the invocation of $S$. Since $t' \geq t$, the GETTIMESTAMP operation invoked by $U$ must have terminated after the GETTIMESTAMP operation invoked by $S$ began. Consequently, $S'$ must have started later than $S$. □

## 6 Consensus

In the consensus problem, processes each start with a private input value and must all choose the same output value. The common output must be the input value of some process. These two conditions are called *agreement* and *validity*, respectively. First, we focus on *binary consensus*, where all inputs are either 0 or 1, and then we solve the more general problem by agreeing on the output bit-by-bit. Herlihy et al. [22] observed that a randomized wait-free consensus algorithm can be "derandomized" to obtain an obstruction-free consensus algorithm. If we derandomize the *anonymous* consensus algorithm of Chandra [14], we obtain the following algorithm.

The algorithm, shown in Fig. 3, uses two unbounded arrays of binary registers, $R_0[1, 2, 3, \ldots]$ and $R_1[1, 2, 3, \ldots]$, all initialized to $\perp$. We use $\bar{v}$ to denote $1 - v$. We refer to the $j$th iteration of the loop as round $j$.

We first give a high-level sketch of how the algorithm works. Each process maintains a preference that is either 0 or 1. Initially, a process's preference is its own input value. A process is said to *change its preference* whenever it executes line 9. Intuitively, the processes are grouped into two teams according to their preference and the teams execute a race along a course of unbounded length that has one track for each preference. Processes mark their progress along the track (which is represented by an unbounded array of binary registers) by changing register values from $\perp$ to $\top$ along the way. Whenever a process $P$ sees that the opposing team is ahead of $P$'s position, $P$ switches its preference to join the other team. As soon as a process observes that it is sufficiently far ahead of all processes on the opposing team, it stops and outputs its own preference. Two processes with opposite preferences could continue to race forever in lockstep but a process running by itself will eventually out-distance all competitors, ensuring obstruction-freedom.

```
PROPOSE(input)
 1  v ← input
 2  j ← 1
 3  loop
 4          if R_v̄[j] = ⊥
 5            then write ⊤ into R_v[j]
 6                 if j > 1 and R_v̄[j − 1] = ⊥
 7                   then return v
 8                 end if
 9            else v ← v̄
10          end if
11          j ← j + 1
12  end loop
```

**Fig. 3** Obstruction-free binary consensus using binary registers

**Theorem 6** *The algorithm in Fig. 3 solves anonymous, obstruction-free binary consensus using binary registers.*

*Proof* We use one key invariant for the correctness proof.
*Invariant*: *For* $v \in \{0, 1\}$, $j \geq 1$, $R_v[j+1] = \top \Rightarrow R_v[j] = \top$.
Consider the first process that writes to $R_v[j + 1]$ in round $j + 1$. In round $j$, that process either wrote to $R_v[j]$, or it switched its preference from $\bar{v}$ to $v$. It can do the latter only if it saw that $R_v[j] \neq \perp$ when it executed line 4 of round $j$. The invariant follows from the fact that entries of $R_v$ are never changed from $\top$ back to $\perp$.
*Obstruction-freedom*: Suppose some process $P$ begins running by itself from some configuration $C$. Let $\hat{j}$ be some value such that $R_0[\hat{j}] = R_1[\hat{j}] = \perp$ in $C$. Eventually, either $P$ will terminate or $P$'s value of $j$ will increase to $\hat{j}$. In the latter case, $P$ will write into either $R_0[\hat{j}]$ or $R_1[\hat{j}]$ and then decide in its next round.
*Validity*: Consider the first process $P$ that changes its preference from $v$ to $\bar{v}$. It did this because it saw that some other process had written into an element of $R_{\bar{v}}$ earlier. That process must have had input $\bar{v}$ (since it did not change its preference before $P$). Thus, if any process ever changes its preference, both input values are present in the execution. The validity condition follows.
*Agreement*: Consider any execution where some process decides. Let $\hat{j}$ be the smallest round in which some process decides. Let $P$ be any process that decides in round $\hat{j}$. Without loss of generality, assume that $P$ decides 0. (The case where $P$ decides 1 is symmetric.) Let $T$ be the time when $P$ last executes the read in line 6. At time $T$, we know that $R_1[\hat{j} - 1] = \perp$. This means that no process could have changed its preference from 0 to 1 during round $\hat{j} - 1$ before time $T$.

We show that no process has begun round $\hat{j}$ before $T$ with preference 1. If there were such a process, it would have entered round $\hat{j}-1$ with preference 1 also, since no process changed its preference from 0 to 1 in round $\hat{j}-1$ before $T$. Therefore, it must have executed line 5 in round $\hat{j}-1$. This contradicts the fact that $R_1[\hat{j}-1]$ is still equal to $\perp$ at time $T$.

Thus, all processes that enter round $\hat{j}$ with preference 1 do so after $T$, so they will all change their preferences to 0 during round $\hat{j}$, and no process can ever write to $R_1[\hat{j}]$ during the entire execution. It follows that each process that enters round $\hat{j}$ with preference 0 will not change its preference during the round, and will either decide 0 or enter the next round with preference 0. Thus, all processes that enter round $\hat{j}+1$ do so with preference 0, and they will all decide 0 during that round. □

The algorithm in Fig. 3 uses an unbounded number of binary registers. We now give a more interesting construction of an obstruction-free, anonymous algorithm for consensus that uses a bounded number of multivalued registers.

Our bounded-space algorithm uses a two-track race course that is circular, with circumference $4n+1$, instead of an unbounded straight one. The course is represented by one array for each track, denoted $R_0[1,2,\ldots,4n+1]$ and $R_1[1,2,\ldots,4n+1]$. We treat these two arrays as a single snapshot object $R$, which we can implement from registers, as described in Proposition 4. Each component stores an integer, initially 0. As a process runs around the race course, it keeps track of which lap it is running. This is incremented each time a process moves from position $4n+1$ to position 1. The progress of processes in the race is recorded by having each process write its lap into the components of $R$ as it passes.

Several complications are introduced by using a circular track. After a fast process records its progress in $R$, a slow teammate who has a smaller lap number could overwrite those values. Although this difficulty cannot be entirely eliminated, we circumvent it with the following strategy. If a process $P$ ever observes that another process is already working on its $k$th lap while $P$ is working on a lower lap, $P$ jumps ahead to the start of lap $k$ and continues racing from there. We use this to ensure that, once there are at least $n+1$ entries containing values greater than or equal to $k$, each process $P$ can later write a lap number lower than $k$ at most once. This limits the amount of overwriting that replaces large lap values with smaller ones. There is a second complication: because some numbers recorded in $R$ may be artificially low due to the overwrites by slow processes, processes may get an incorrect impression of which team is in the lead. To handle this, we make processes less fickle: they switch teams only when they have lots of evidence that the other team is in the lead. Also, we require a process to have evidence that it is leading by a very wide margin before it decides. The algorithm is given in Fig. 4, where we use $\bar{v}$ to denote $1-v$.

```
PROPOSE(input)
 1  v ← input
 2  j ← 0
 3  lap ← 1
 4  loop
 5          S ← SCAN of R
 6          if S_v[i] < S_v̄[i] for a majority of values of
                  i ∈ {1,..,4n+1}
 7            then v ← v̄
 8          end if
 9          if min_{1≤i≤4n+1} S_v[i] > max_{1≤i≤4n+1} S_v̄[i]
10            then return v
11          elseif some element of S is greater than lap
12            then lap ← maximum element of S
13                 j ← 1
14          else   j ← j+1
15                 if j = 4n+2
16                   then lap ← lap+1
17                        j ← 1
18                 end if
19          end if
20          UPDATE the value of R_v[j] to lap
21  end loop
```

**Fig. 4** Obstruction-free consensus using $O(n)$ registers

**Theorem 7** *The algorithm in Fig. 4 is an anonymous, obstruction-free binary consensus algorithm that uses* $8n+2$ *registers.*

*Proof* We use $8n+2$ registers to get a non-blocking implementation of the snapshot object $R$ using Proposition 4. We now prove the obstruction-freedom, validity and agreement properties.

*Obstruction-freedom*: Consider any configuration $C$. Let $m$ be the maximum value that appears in any component of $R$ in $C$. Suppose some process $P$ runs by itself forever without halting, starting from $C$. It is easy to check that $P$'s local variable *lap* increases at least once every $4n+1$ iterations of the loop. Eventually $P$ will have $lap \geq m+1$ and $j=1$. Let $v_0$ be $P$'s local value of $v$ when $P$ next executes line 9. At this point, no entries in $R$ are larger than $m$. Furthermore, $R_{v_0}[i] \geq R_{\bar{v}_0}[i]$ for a majority of the values $i$; otherwise $P$ would have changed its value of $v$ in the previous iteration. From this point onward, $P$ will never change its local value $v$, since it will write only values bigger than $m$ to $R_{v_0}$, and $R_{\bar{v}_0}$ contains no elements larger than $m$, so none of $P$'s future writes will ever make the condition in line 6 true. During the next $4n+1$ iterations of the loop, $P$ will write its value of *lap* into each of the entries of $R_{v_0}$, and then the termination condition will be satisfied, contrary to the assumption that $P$ runs forever. (This termination occurs within $O(n)$ iterations of the loop, once $P$ has started to run on its own, so termination is guaranteed as soon as any process takes $O(n^4)$ steps by itself, since the SCAN algorithm of Proposition 4 terminates if a process takes $O(n^3)$ steps by itself.)

*Validity*: Suppose all processes begin with input 0. (The case when they all start with input 1 is symmetric.) Then we have the following invariants:

(1) $R_1[i] = 0$ for all $i$, and
(2) Every process's local variable $v$ is equal to 0.

These are easy to prove: if they are true, the test in line 6 will always fail, so (2) can never become false, and this means that no process can every write to $R_1$ in line 20. Thus any process that decides in line 10 can only decide 0.

*Agreement*: For each process that decides, consider the moment when it last performs a SCAN of $R$. Let $T$ be the first such moment in the execution. Let $S^*$ be the SCAN taken at time $T$. Without loss of generality, assume the value decided by the process that did this SCAN is 0. We shall show that every other process that terminates also decides 0. Let $m$ be the minimum value that appears in $S_0^*$. Note that all values in $S_1^*$ are less than $m$.

We first show that, after $T$, at most $n$ UPDATES write a value smaller than $m$ into $R$. If not, consider the first $n + 1$ such UPDATES after $T$. At least two of them are done by the same process, say $P$. Process $P$ must do a SCAN in between the two UPDATES. That SCAN would still see one of the values in $R_0$ that is at least $m$, since $4n + 1 > n$. Immediately after this SCAN, $P$ would change its local variable *lap* to be at least $m$ and the value of *lap* is non-decreasing, so $P$ could never perform the second UPDATE with a value smaller than $m$.

We use a similar argument to show that, after $T$, at most $n$ UPDATE operations write a value into $R_1$. If this is not the case, consider the first $n + 1$ such UPDATES after $T$. At least two of them are performed by the same process, say $P$. Process $P$ must do a SCAN between the two UPDATES. Consider the last SCAN that $P$ does between these two UPDATES. That SCAN will see at most $n$ values in $R_1$ that are greater than or equal to $m$, since all such values were written into $R_1$ after $T$. It will also see at most $n$ values in $R_0$ that are less than $m$ (by the argument in the previous paragraph). Thus, there will be at least $2n + 1$ values of $i$ for which $R_0[i] \geq m > R_1[i]$ when the SCAN occurs. Thus, immediately after the SCAN, $P$ will change its local value of $v$ to 0 in line 7, contradicting the fact that it writes into $R_1$ later in that iteration.

It follows from the preceding two paragraphs that, at all times after $T$, $\min_{1 \leq i \leq 4n+1} R_1[i] < m \leq \max_{1 \leq i \leq 4n+1} R_0[i]$. Any process that takes its final SCAN after $T$ cannot decide 1. □

Just as a randomized, wait-free consensus algorithm can be "derandomized" to yield an obstruction-free algorithm, the algorithm in Fig. 4 could be used as the basis of a randomized wait-free anonymous algorithm that solves binary consensus using bounded space.

Theorems 6 and 7 can be extended to non-binary consensus using the following proposition, which is proved by constructing a consensus algorithm where processes agree on the output bit-by-bit.

**Proposition 8** *If there is an anonymous, obstruction-free algorithm for binary consensus using a set of objects S, then there is an anonymous, obstruction-free algorithm for consensus with inputs from the countable set D that uses* $|D|$ *binary registers and* $\log |D|$ *copies of S. Such an algorithm can also be implemented using* $2 \log |D|$ *registers and* $\log |D|$ *copies of S if* $|D|$ *is finite.*

*Proof* The standard way to solve multi-valued consensus using binary consensus is to agree on each bit of the output using a separate instance of binary consensus. (If $D$ is countably infinite, we can encode each element as a finite string using the alphabet {00, 01, 10} and use 11 to mark the end of a value so that processes will know when they can stop agreeing on bits and decide.)

We first describe how to solve multi-valued consensus using an additional set of $|D|$ binary registers, $\{R[v] : v \in D\}$, that are initialized to $\perp$. Before taking any steps, a process with input $v$ writes $\top$ into $R[v]$. Assume that all process have agreed upon the first $m - 1$ bits of the output, $b_1, \ldots, b_{m-1}$. Assume also that every process stores (locally) a preference that is the input of some process and begins with $b_1 b_2 \ldots b_{m-1}$. Processes execute binary consensus, using the $m$th bit of their preferred value as the input, to decide on the next bit of the output, $b_m$. If the value agreed upon is different from the value a process proposed, that process must change its preference to one that agrees with the first $m$ bits that have been decided. It can do this by searching the array $R$ for such a $v$ that has $R(v) = \top$.

If $D$ is finite, we can use a similar construction that has $2 \log |D|$ additional registers instead of the binary registers of $R$. Let the additional registers be $R_0[1 \ldots \log |D|]$ and $R_1[1 \ldots \log |D|]$. Before a process proposes a value $b$ to the $m$th instance of binary consensus, which is being used to determine the $m$th bit of the output, it writes its preference into $R_b[m]$. (Initially, a process's preference is its own input value.) If a process must change its preference as a result of the binary consensus, it uses the value it then reads from $R_{\bar{b}}[m]$. □

The following corollaries follow directly from Theorems 6 and 7 and the preceding proposition.

**Corollary 9** *There is an anonymous, obstruction-free algorithm for consensus, with arbitrary inputs, using binary registers. There is an anonymous, obstruction-free algorithm for consensus with inputs from a finite set D that uses* $(8n + 4) \log |D|$ *registers.*

## 7 Obstruction-free implementations

We now give a complete characterization of the (deterministic) object types that have anonymous, obstruction-free implementations from registers. We say that an object is

idempotent if, starting from any state, two successive invocations of the same operation, with the same arguments, return the same response and leave the object in a state that is indistinguishable from the state a single application would leave it in. (This is a slightly more general definition of idempotence than the one used in [5].) This definition of idempotence is made more precise using the formalism of Aspnes and Herlihy [6]. A *sequential history* is a sequence of steps applied to a particular object, each step being a pair consisting of an operation invocation and its response. Such a history is called *legal* (for a given initial state) if it is consistent with the specification of the object's type.

**Definition 10** ([6]) Two sequential histories $H$ and $H'$ are *equivalent* if, for all sequential histories $G$, $H \cdot G$ is legal if and only if $H' \cdot G$ is legal.

**Definition 11** A step $p$ is *idempotent* if, for all sequential histories $H$, if $H \cdot p$ is legal then $H \cdot p \cdot p$ is legal and equivalent to $H \cdot p$.

An object is called idempotent if all of its operations are idempotent. Examples of idempotent objects include registers, sticky bits, snapshot objects and resettable consensus objects.

**Theorem 12** *A deterministic object type $T$ has an anonymous, obstruction-free implementation from binary registers if and only if $T$ is idempotent.*

*Proof* ($\Rightarrow$) To derive a contradiction, assume that there is an implementation of $T$, but $T$ is not idempotent. Let $H = h_1 \cdot h_2 \cdots h_k$ be a minimal length history that violates the idempotence property. That is, there is a step $p$ such that $H \cdot p$ is legal, but either $H \cdot p \cdot p$ is not legal or $H \cdot p$ and $H \cdot p \cdot p$ are not equivalent.

Let $H_i = h_1 \cdot h_2 \cdots h_i$ be the prefix consisting of the first $i$ steps of $H$. Let $H'_i = h_1 \cdot h_1 \cdot h_2 \cdot h_2 \cdots h_i \cdot h_i$ and let $H' = H'_k$. We prove by induction on $i$ that $H'_i$ is legal and equivalent to $H_i$ for $0 \leq i \leq k$. The base case, when $i = 0$, is trivial, since both histories are empty. Assume the claim is true for $i - 1$. It follows that $H'_{i-1} \cdot h_i \cdot h_i = H'_i$ is legal and equivalent to $H_{i-1} \cdot h_i \cdot h_i$. The history $H_{i-1} \cdot h_i \cdot h_i$ is legal and equivalent to $H_{i-1} \cdot h_i = H_i$, by the minimality of $H$. By the transitivity of equivalence, $H'_i$ is equivalent to $H_i$, completing the proof of the claim. Thus, $H'$ is equivalent to $H$.

Let $P$ and $Q$ be two distinct processes. Consider the execution $\alpha$ in which process $P$ executes the implementation's code for the sequence of operations in $H \cdot p$. (Since the type $T$ is deterministic, the sequence of responses $P$ gets for all operations will match the ones that appear in the history $H \cdot p$.) Let $\beta$ be the execution where $P$ and $Q$ execute the implementation's code for the sequence of operations in $H \cdot p$, taking alternate steps. Since $P$ and $Q$ access only registers, they will take exactly the same sequence of steps, and both will generate identical responses for each operation. The execution $\beta$ therefore simulates the history $H' \cdot p \cdot p$, and this history must be legal; otherwise the implementation would be incorrect. Since $H'$ is equivalent to $H$, the history $H \cdot p \cdot p$ must also be legal.

The internal state of $P$ is the same at the end of $\alpha$ and $\beta$. The value stored in each register is also the same at the end of these two runs. Thus, any sequence of operations simulated by $P$ after $\alpha$ will generate exactly the same sequence of responses as they would if $P$ simulated them after $\beta$. It follows that for any history $G$, $H \cdot p \cdot G$ is legal if and only if $H' \cdot p \cdot p \cdot G$ is legal. Thus, $H \cdot p$ is equivalent to $H' \cdot p \cdot p$, which is equivalent to $H \cdot p \cdot p$ (since $H'$ is equivalent to $H$). This contradicts the definition of $H$.

($\Leftarrow$) Let $T$ be any idempotent type. We give an anonymous, obstruction-free algorithm that implements $T$ from binary registers. The algorithm uses an unbounded number of consensus objects $Con[1, 2, \ldots]$, which have an obstruction-free implementation from binary registers, by Corollary 9. The algorithm also uses the GETTIMESTAMP operation that accesses a weak counter, which can also be implemented from binary registers, according to Theorem 3. These will be used to agree on the sequence of operations performed on the simulated object. All other variables are local. The *history* variable is initialized to an empty sequence, and $i$ is initialized to 1. The code in Fig. 5 describes how a process simulates an operation $op$.

```
DO(op)
 1  loop
 2        t ← GETTIMESTAMP
 3        (op', t') ← PROPOSE(op, t) to Con[i]
 4        res ← result returned to op' if it is done
                after history
 5        history ← history · (op, res)
 6        i ← i + 1
 7        if (op', t') = (op, t)
 8          then return res
 9        end if
10  end loop
```

**Fig. 5** Obstruction-free implementation of an idempotent object from binary registers

*Obstruction-freedom*: If, after some point of time, only one process takes steps, all of its subroutine calls will terminate, and it will eventually increase $i$ until it accesses a consensus object that no other process has accessed. When that happens, the loop is guaranteed to terminate.

*Linearizability*: We must describe how to linearize all of the simulated operations. Any simulated operation that receives a result in line 3 that is equal to the value it proposed to the consensus object is linearized at the moment that consensus object was first accessed. All (identical) operations linearized at the moment $Con[i]$ is first accessed are said to belong to group $i$.

The following invariant follows easily from the code (and the fact that the object is idempotent): At the beginning of any iteration of the loop by any process $P$, $history_P$ is equivalent to the history that would result from the first $i_P - 1$ groups of simulated operations taking place (in order), where $i_P$ and $history_P$ are $P$'s local values of the variables $i$ and $history$. Thus, the results returned to all simulated operations are consistent with the linearization.

We must still show that the linearization point chosen for a simulated operation is between its invocation and response. Let $D$ be an execution of Do($op$) in group $i$. The linearization point $T$ of $D$ is the first access in the execution to $Con[i]$. Clearly, this cannot be after $D$ completes, since $D$ itself accesses $Con[i]$. Let $D'$ be the execution of Do($op'$) that first accesses $Con[i]$. (It is possible that $D = D'$.) Since $D$ is linearized in group $i$, it must be the case that $op = op'$, and also that the timestamps used in the proposals by $D$ and $D'$ to $Con[i]$ are equal. Let $t$ be the value of this common timestamp. Note that $T$ occurs after $D'$ has completed the GetTimestamp operation that returned $t$. If $T$ were before $D$ is invoked, then the GetTimestamp operation that $D$ calls would have to return a timestamp larger than $t$. Thus, $T$ is after the invocation of $D$, as required. □

The algorithm used in the above proof does not require processes to have knowledge of the number of processes, $n$, so the characterization of Theorem 12 applies whether or not processes know $n$. Since unbounded registers are idempotent, it follows from the theorem that they have an obstruction-free implementation from binary registers, and we get the following corollary.

**Corollary 13** *An object type $T$ has an anonymous, obstruction-free implementation from unbounded registers if and only if $T$ is idempotent.*

In the more often-studied context of non-anonymous wait-free computing, counters (with separate increment and read operations) can be implemented from registers [6], while consensus objects cannot be [21,29]. The reverse is true for anonymous, obstruction-free implementations (since consensus is idempotent, but counters are not). Thus, the traditional classification of object types according to their consensus numbers [21] will not tell us very much about anonymous, obstruction-free implementations since, for example, consensus objects cannot implement counters, which have consensus number 1.

If large registers are available (instead of just binary registers), the algorithm in Fig. 5 could use, as a consensus subroutine, the algorithm of Theorem 7 instead of the algorithm of Theorem 6. If the number of different operations that are permitted on the idempotent object type is $d$ and $k$ invocations occur, then the number of registers needed to implement each consensus object is $O(n \log(dk))$, by Proposition 8, and at most $k$ consensus objects are needed. This yields the following proposition.

**Proposition 14** *An idempotent object with a operation set of size $d$ has an implementation that uses $O(kn \log(dk))$ registers in any execution with $k$ invocations on the object.*

## 8 Concluding remarks

This paper is the first investigation of what object types can be deterministically implemented in an anonymous, asynchronous shared-memory system. In particular, we gave a characterization of the class of objects with obstruction-free implementations. A number of related questions are open.

Determining the class of objects that have wait-free implementations is a challenging objective. It will be strictly smaller than the class of objects with obstruction-free implementations, since consensus has no wait-free solution, but how much smaller?

Can binary registers implement multi-valued registers in a wait-free way? In this paper, we assumed atomic registers. It is known that such registers can be obtained from binary safe registers assuming identities. Is this also the case in an anonymous system?

Many of the implementations in this paper used either unbounded registers or an unbounded number of binary registers. Does anonymity inherently require unbounded space to solve some problems?

For some of the problems in this paper, subsequent work has reduced the number of (unbounded) registers required. Ellen et al. [16] gave an anonymous wait-free weak counter implementation that uses $n$ registers. They also gave a bounded wait-free weak counter algorithm using $O(n^2)$ registers. Plugging the former weak counter into the snapshot algorithm of Theorem 5 in this paper yields a wait-free snapshot implementation that uses $m + n$ registers for $m$ components. Plugging in the latter timestamp algorithm instead gives an anonymous snapshot implementation from $O(m + n^2)$ registers, and this is the first bounded wait-free anonymous algorithm for the snapshot problem. The same authors proved a time-space tradeoff for anonymous implementations of snapshots: if an implementation uses $r$ registers, $\Omega(n/r)$ steps are needed to perform a Scan, even if $m = 2$ [18].

This paper focused on implementations from registers. A sequel to this paper studies the possibility of wait-free algorithms for consensus and naming problems in anonymous systems equipped with stronger types of shared objects [33].

**Acknowledgments** We thank Petr Kouznetsov for helpful conversations and the referees (who were, fittingly, anonymous) for their comments. This research was supported by the Swiss National Science Foundation (NCCR MICS project) and the Natural Sciences and

Engineering Research Council of Canada. A preliminary version of this paper appeared in [20].

## References


1. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. J. ACM **40**(4), 873–890 (1993)
2. Anderson, J.H.: Composite registers. Distrib. Comput. **6**(3), 141–154 (1993)
3. Angluin, D.: Local and global properties in networks of processors. In: Proceedings of 12th ACM Symposium on Theory of Computing, pp. 82–93 (1980)
4. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. Distrib. Comput. **18**(4), 235–253 (2006)
5. Aspnes, J., Fich, F.E., Ruppert, E.: Relationships between broadcast and shared memory in reliable anonymous distributed systems. Distrib. Comput. **18**(3), 209–219 (2006)
6. Aspnes, J., Herlihy, M.: Wait-free data structures in the asynchronous PRAM model. In: Proceedings of 2nd ACM Symposium on Parallel Algorithms and Architectures, pp. 340–349 (1990)
7. Aspnes, J., Shah, G., Shah, J.: Wait-free consensus with infinite arrivals. In: Proceedings of 34th ACM Symposium on Theory of Computing, pp. 524–533 (2002)
8. Attiya, H., Gorbach, A., Moran, S.: Computing in totally anonymous asynchronous shared memory systems. Inf. Comput. **173**(2), 162–183 (2002)
9. Attiya, H., Guerraoui, R., Kouznetsov, P.: Computing with reads and writes in the absence of step contention. In: Distributed Computing, 19th International Conference, vol. 3724 of *LNCS*, pp. 122–136 (2005)
10. Bazzi, R.A., Ding, Y.: Non-skipping timestamps for Byzantine data storage systems. In: Distributed Computing, 18th International Conference, vol. 3274 of *LNCS*, pp. 405–419 (2004)
11. Berthold, O., Federrath, H., Köhntopp, M.: Project "anonymity and unobservability in the internet". In: Proceedings of 10th Conference on Computers, Freedom and Privacy, pp. 57–65 (2000)
12. Bono, S.C., Soghoian, C.A., Monrose, F.: Mantis: A lightweight, server-anonymity preserving, searchable P2P network. Technical Report TR-2004-01-B-ISI-JHU, Information Security Institute, Johns Hopkins University (2004)
13. Buhrman, H., Panconesi, A., Silvestri, R., Vitanyi, P.: On the importance of having an identity or, is consensus really universal?. Distrib. Comput. **18**(3), 167–176 (2006)
14. Chandra, T.D.: Polylog randomized wait-free consensus. In: Proceedings of 15th ACM Symposium on Principles of Distributed Computing, pp. 166–175 (1996)
15. Drulă, C.: The totally anonymous shared memory model in which the number of processes is known. Personal communication
16. Ellen, F., Fatourou, P., Ruppert, E.: The space complexity of unbounded timestamps. In: Proceedings of 21st International Symposium on Distributed Computing (2007, to appear)
17. Eğecioğlu, O., Singh, A.K.: Naming symmetric processes using shared variables. Distrib. Comput. **8**(1), 19–38 (1994)
18. Fatourou, P., Fich, F.E., Ruppert, E.: Time-space tradeoffs for implementations of snapshots. In: Proceedings of 38th ACM Symposium on Theory of Computing (2006)
19. Goldschlag, D., Reed, M., Syverson, P.: Onion routing. Commun. ACM **42**(2), 39–41 (1999)
20. Guerraoui, R., Ruppert, E.: What can be implemented anonymously? In: Distributed Computing, 19th International Conference, vol. 3724 of *LNCS*, pp. 244–259 (2005)
21. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. **13**(1), 124–149 (1991)
22. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: double-ended queues as an example. In: Proceedings of 23rd IEEE International Conference on Distributed Computing Systems, pp. 522–529 (2003)
23. Herlihy, M., Shavit, N.: The topological structure of asynchronous computability. J. ACM **46**(6), 858–923 (1999)
24. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
25. Jayanti, P., Toueg, S.: Wakeup under read/write atomicity. In: Distributed Algorithms, 4th International Workshop, vol. 486 of *LNCS*, pp. 277–288 (1990)
26. Johnson, R.E., Schneider, F.B.: Symmetry and similarity in distributed systems. In: Proceedings of 4th ACM Symposium on Principles of Distributed Computing, pp. 13–22 (1985)
27. Kutten, S., Ostrovsky, R., Patt-Shamir, B.: The Las-Vegas processor identity problem (How and when to be unique). J. Algorithms **37**(2), 468–494 (2000)
28. Lipton, R.J., Park, A.: The processor identity problem. Inf. Process. Lett. **36**(2), 91–94 (1990)
29. Loui, M.C., Abu-Amara, H.H.: Memory requirements for agreement among unreliable asynchronous processes. In: Preparata, F.P. (ed.) Advances in Computing Research, vol. 4., pp. 163–183. JAI Press, Greenwich (1987)
30. Neiger, G.: Set-linearizability. In: Proceedings of 13th ACM Symposium on Principles of Distributed Computing, pp. 396 (1994)
31. Panconesi, A., Papatriantafilou, M., Tsigas, P., Vitányi, P.: Randomized naming using wait-free shared variables. Distrib. Comput. **11**(3), 113–124 (1998)
32. Reiter, M.K., Rubin, A.D.: Crowds: anonymity for web transactions. ACM Trans. Inf. Syst. Secur. **1**(1), 66–92 (1998)
33. Ruppert, E.: The anonymous consensus hierarchy and naming problems. Technical Report CSE-2006-11, Department of Computer Science and Engineering, York University (2006)
34. Teng, S.-H.: Space efficient processor identity protocol. Inf. Process. Lett. **34**(3), 147–154 (1990)