What Can Be Implemented Anonymously?

Rachid Guerraoui[1] and Eric Ruppert[2]

[1] EPFL, Lausanne, Switzerland
[2] York University, Toronto, Canada

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: timestamping, 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.

1 Introduction

Distributed computing typically studies what can be computed by a system of n processes that can fail independently. Variations on the capacities of the processes (e.g., in terms of memory or time), their means of communication (e.g., shared memory or message passing), and their failure modes (e.g., crash failures or malicious failures) 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).

Several properties have been defined to describe the progress made by an algorithm regardless of process crashes or asynchrony. The strongest is *wait-freedom* [18], which requires *every* non-faulty process to complete its algorithm in a finite number of its own steps. However, wait-free algorithms are often 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* [19], 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.

Virtually all of the literature on those topics assumes that processes have distinct identities. Besides intellectual curiosity, it is practically appealing to revisit this fundamental assumption. Indeed, certain systems, like sensor networks, consist of mass-produced tiny agents that might not even have identifiers [4].

Others, like web servers [29] and peer-to-peer file sharing systems [11], sometimes mandate preserving the anonymity of the users and forbid the use of any form of identity for the sake of privacy [10]. Instead of revealing its identity to a server that houses a shared-memory object, a process might use a trusted third party that can itself be approximated by a decentralized mechanism [16]. This party forwards the process's invocations to the server (stripped of the process's id) and then forwards the server's responses back to the process. But what can actually be done in an *anonymous* system? In such a system, processes are programmed identically [5, 7, 8, 12, 23, 28]. In particular, processes do not have identifiers. There has been work on anonymous message-passing systems, starting with Angluin [3]. The very 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.)

We explore in this paper, for the first time, 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. Some problems, such as leader election, are clearly impossible in this model 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 are traditionally used for creating timestamps, cannot be implemented in our model, so 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 an 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 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 [18]. 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 im-

Theorem	Implemented Object	Using	Space	Progress	Uses
1	weak counter	registers	$O(k)$	wait-free	
3	weak counter	binary registers	$O(k)$	non-blocking	
4	m-component snapshot	registers	m	non-blocking	
5	m-component snapshot	registers	$O(m+k)$	wait-free	1
6	binary consensus	binary registers	unbounded	obs-free	
7	binary consensus	registers	$O(n)$	obs-free	4
9	consensus	binary registers	unbounded	obs-free	6, 8
9	consensus	registers	$O(n \log d)$	obs-free	7, 8
10	idempotent object	binary registers	unbounded	obs-free	3, 9
12	idempotent object	registers	object-dependent	obs-free	3, 7

Fig. 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.

plementation [19]. 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 [18, 26]. In Sect. 6, we note that an obstruction-free anonymous consensus algorithm can be obtained by simply derandomizing the randomized anonymous algorithm of Chandra [13]. 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. We use a symmetry argument to show this condition is necessary. 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 summarizes all anonymous implementations given in this paper, indicating which implementations are used as subroutines for others.

2 Related Work

Some research has studied anonymous shared-memory systems when no failures can occur. Johnson and Schneider [23] gave leader election algorithms using versions of single-writer snapshots and test&set objects. Attiya, Gorbach and Moran [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 [14]. Consensus is solvable in these models, but it is not solvable if the registers cannot be initialized by the programmer [22]. Aspnes, Fich and Ruppert [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.

There has also been some research on randomized algorithms for anonymous shared-memory systems with no failures. For the naming problem, processes must choose unique names for themselves. 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 [25, 30]. Two papers gave randomized renaming algorithms that have finite expected running time, and hence terminate with probability 1 [15, 24].

Randomized algorithms for systems with crash failures have also been studied. Panconesi *et al.* [28] 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 [12, 15, 24]. Interestingly, Buhrman *et al.* [12] gave a randomized wait-free anonymous algorithm for consensus in this model that is based on Chandra's randomized consensus algorithm [13]. Thus, producing unique identifiers is strictly harder than consensus in the randomized setting. Aspnes, Shah and Shah [7] extended the 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 [20] gave a characterization of the decision tasks that have wait-free solutions in non-anonymous systems using ideas borrowed from algebraic topology. They also describe how the characterization can be extended to systems with a kind of anonymity: processes have identifiers but are only allowed to use them in very limited ways. Herlihy gave a *universal construction* which describes how to create a wait-free implementation of any object type using consensus objects [18]. 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 [9] introduced, in the context of Byzantine systems, non-skipping timestamps, a stronger abstraction than what we call a weak counter. (Our weak counter does not preclude skipping values.)

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) [21]: 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 [27].) 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 are somewhat inconsistent with the notion of totally anonymous systems, since processes must identify themselves when they invoke an operation.)

Some types of objects are provided by the system and all other types needed 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.

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.) Furthermore, the value returned to any operation should not exceed the number of invocations that have occurred so far. 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 requirement that all values returned should be distinct. It is known that a fetch&increment object has no wait-free implementation from registers, even if processes have identifiers [18]. By considering our weaker version, we have an object that is implementable, and still strong enough for our purposes.

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.

```
GetTimestamp
 1  $b \leftarrow a + 1$
 2  $\ell \leftarrow L$; $t \leftarrow \ell$; $j \leftarrow 0$
 3  loop until $A[b] = \bot$
 4          if $L \neq \ell$
 5            then $\ell \leftarrow L$; $t \leftarrow \max(t, \ell)$; $j \leftarrow j + 1$
 6                  if $j \geq n$ then $a \leftarrow b + 1$; return $t$ and halt
 7                  end if
 8          end if
 9          $b \leftarrow 2b - a + 1$
10  end loop
11  loop until $a = b$
12          $mid \leftarrow \frac{a+b-1}{2}$        ▷ This is an integer, since $b - a + 1$ is a power of 2
13          if $A[mid] = \bot$ then $b \leftarrow mid$
14          else $a \leftarrow mid + 1$
15          end if
16  end loop
17  write $\top$ to $A[b]$
18  $L \leftarrow b$
19  return $b$
```

Fig. 2. Wait-free implementation of a weak counter from registers.

Our wait-free implementation uses an array $A[1, 2, \ldots]$ of binary registers, each initialized to $\bot$. To obtain a counter value, a process locates the first entry of the array that is $\bot$, changes it to $\top$, and returns the index of this entry. (See Fig. 2.) The key property for correctness is the following invariant: if $A[k] = \top$, then all entries in $A[1..k]$ are $\top$. To locate the first $\bot$ 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 $\bot$ in 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 $A[a..b]$ 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. *Fig. 2 gives a wait-free, anonymous 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 14 maintains Invariant 1. Line 6 does too, since the preceding

iteration of line 3 found that $A[b] = \top$.
Invariant 2: If $A[k] = \top$, then $A[k'] = \top$ for all $k' \leq k$.
This follows from Invariant 1: whenever line 17 is executed, we have $a = b$, so $A[b-1]$ is already $\top$.
Invariant 3: Whenever a process P executes line 11 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 easy to prove by induction on the number of iterations of the second loop.

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: 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 6, 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 19, it is also clear that $A[r_1] = \top$ when op_1 terminates.

If op_2 terminates in line 19, 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 6, 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. Thus, 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 19, 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 $r_2 \geq r_3 > r_1$. ■

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 $O(\log k)$. An amortized analysis can be used 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. (See [17] for a detailed proof of the following.)

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

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, 4–8 and 18, 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.

```
UPDATE($i, x$)
$t \leftarrow$ GETTIMESTAMP
$v \leftarrow$ SCAN
write $(x, v, t)$ in $R_i$
```

```
SCAN
$t \leftarrow$ GETTIMESTAMP
loop
        read $R_1, R_2, \ldots, R_m$
        if a register contained $(*, v, t')$ with $t' \geq t$
            then return $v$
        elseif $n + 1$ sets of reads gave same results
            then return the first field of each value
        end if
end loop
```

Fig. 3. Wait-free implementation of a snapshot object from registers.

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. Many algorithms exist to implement snapshots, but all use process identifiers. The following proposition can be proved using a simple modification of the standard non-blocking snapshot algorithm for non-anonymous systems [1]. A proof appears in [17].

Proposition 4. *There is a non-blocking, anonymous implementation of an m-component snapshot object from m registers.*

More surprisingly, we show that a standard algorithm for (non-anonymous) wait-free snapshots [1] can also be modified to work in an anonymous system. 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. See Fig. 3.

Theorem 5. *The algorithm in Fig. 3 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. (Sketch) See [17] for a detailed proof. It can be shown that the registers either keep changing continually, eventually including timestamps that will satisfy the first termination condition, or stop changing so that the second termination condition will eventually be satisfied. UPDATES are linearized when the write occurs. If a SCAN sees $n+1$ identical sets of reads, it can be shown that these values were all in the register at one instant in time, which is used as the linearization point. If a SCAN uses the vector recorded from another SCAN as its output, the two SCANS are linearized at the same time. The timestamp mechanism is sufficient to guarantee that the linearization point so chosen is between the invocation and response of the SCAN. ■

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. Herlihy, Luchangco and Moir [19] 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 [13], we obtain the following theorem. (A proof appears in [17].)

Theorem 6. *There is an anonymous, obstruction-free binary consensus algorithm using binary registers.*

The construction that proves Theorem 6 uses an unbounded number of binary registers. In this section, we give a more interesting construction of an obstruction-free, anonymous algorithm for consensus that uses a bounded number of (multivalued) registers. First, we focus on *binary consensus*, where all inputs are either 0 or 1, and give an algorithm using $O(n)$ registers.

In the unbounded-space algorithm, each process maintains a preference that is either 0 or 1. Initially, a process's preference is its own input value. 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.

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. 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 $4n+1$

```
PROPOSE(input)
 1  $v \leftarrow input$; $j \leftarrow 0$; $lap \leftarrow 1$
 2  loop
 3        $S \leftarrow$ SCAN of $R$
 4        if $S_v[i] < S_{\bar{v}}[i]$ for a majority of values of $i \in \{1, .., 4n+1\}$
 5          then $v \leftarrow \bar{v}$
 6        end if
 7        if $\min_{1 \le i \le 4n+1} S_v[i] > \max_{1 \le i \le 4n+1} S_{\bar{v}}[i]$
 8          then return $v$
 9        elseif some element of $S$ is greater than $lap$
10          then $lap \leftarrow$ maximum element of $S$; $j \leftarrow 1$
11        else     $j \leftarrow j + 1$
12                 if $j = 4n+2$ then $lap \leftarrow lap + 1$; $j \leftarrow 1$
13                 end if
14        end if
15        UPDATE the value of $R_v[j]$ to $lap$
16  end loop
```

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

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 eliminated, we circumvent it with the following strategy. If a process P ever observes that another process is already working on its kth lap while P is working on a lower lap, P jumps ahead to the start of lap k and continues racing from there. This will ensure that P can only overwrite one location with a lower lap number, once sufficiently many k's have been written. 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 $1 - v$.

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.

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 until P decides. 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 7. 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 step.) 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 4 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: If all processes start with the same input value v, they will never switch to preference $\bar{v}$ nor write into any component of $R_{\bar{v}}$.

Agreement: For each process that decides, consider the moment when it last scans 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 proof 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 5, 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, $\min\limits_{1 \leq i \leq 4n+1} R_1[i] < m \leq \max\limits_{1 \leq i \leq 4n+1} R_0[i]$. 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 of Theorem 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 using a fairly standard technique of agreeing on the output bit-by-bit (see [17]).

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.*

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

Proof. ($\Rightarrow$) We assume $n > 2$. The special case $n = 2$ is deferred to the full paper. Assume there is such an implementation of T. Let P, Q and R be distinct processes. Let H be any legal history and let $p = (op, res)$ be any step such that $H \cdot p$ is legal. Let α be the execution of the implementation where some process P executes the code for the sequence of operations in H, and then Q executes *op*. Since the object is deterministic, Q must receive the result res for operation *op*. Let β be the execution where P executes the code for the sequence of operations in H, and then processes Q and R execute the code for *op*, taking alternate steps. Since Q and R access only registers, they will take exactly the same sequence of steps, and both will terminate and return res. Thus, $H \cdot p \cdot p$ must be legal also.

The internal state of P is the same at the end of α and β. The value stored in each register is also the same at the end of these two runs. Thus any sequence of operations performed by P after α will generate exactly the same sequence of responses as they would if P executed them after β. 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, so T is idempotent.

```
Do($op$)
1  loop
2      $t \leftarrow$ GetTimestamp
3      $(op', t') \leftarrow$ Propose($op, t$) to $Con[i]$
4      $res \leftarrow$ result returned to $op'$ if it is done after $history$
5      $history \leftarrow history \cdot (op, res)$
6      $i \leftarrow i + 1$
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.

($\Leftarrow$) 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*.

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.

Correctness: 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 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 10 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 11. *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 [18, 26]. 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 [18] 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 12. *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.*

Acknowledgements We thank Petr Kouznetsov for helpful conversations. This research was supported by the Swiss National Science Foundation (NCCR MICS project) and the Natural Sciences and Engineering Research Council of Canada.

References

1. Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.
2. J. H. Anderson. Composite registers. *Distributed Computing*, 6(3):141–154, 1993.
3. D. Angluin. Local and global properties in networks of processors. In *12th ACM Symp. on Theory of Computing*, pages 82–93, 1980.
4. D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. In *23rd ACM Symp. on PODC*, pages 290–299, 2004.
5. J. Aspnes, F. Fich, and E. Ruppert. Relationships between broadcast and shared memory in reliable anonymous distributed systems. In *Distributed Computing, 18th Intl Symp.*, pages 260–274, 2004.

6. J. Aspnes and M. Herlihy. Wait-free data structures in the asynchronous PRAM model. In *2nd ACM SPAA*, pages 340–349, 1990.
7. J. Aspnes, G. Shah, and J. Shah. Wait-free consensus with infinite arrivals. In *34th ACM Symp. on Theory of Computing*, pages 524–533, 2002.
8. H. Attiya, A. Gorbach, and S. Moran. Computing in totally anonymous asynchronous shared memory systems. *Inf. and Computation*, 173(2):162–183, 2002.
9. R. A. Bazzi and Y. Ding. Non-skipping timestamps for byzantine data storage systems. In *Distributed Computing, 18th Intl Conf.*, pages 405–419, 2004.
10. O. Berthold, H. Federrath, and M. Köhntopp. Project "anonymity and unobservability in the internet". In *10th Conf. on Computers, Freedom and Privacy*, pages 57–65, 2000.
11. S. C. Bono, C. A. Soghoian, and F. Monrose. Mantis: A lightweight, server-anonymity preserving, searchable P2P network. Technical Report TR-2004-01-B-ISI-JHU, Information Security Institute, Johns Hopkins University, 2004.
12. H. Buhrman, A. Panconesi, R. Silvestri, and P. Vitanyi. On the importance of having an identity or, is consensus really universal? In *Distributed Computing, 14th Intl Conf.*, volume 1914 of *LNCS*, pages 134–148, 2000.
13. T. D. Chandra. Polylog randomized wait-free consensus. In *15th ACM Symp. on PODC*, pages 166–175, 1996.
14. C. Drulă. The totally anonymous shared memory model in which the number of proces ses is known. Personal communication.
15. O. Eğecioğlu and A. K. Singh. Naming symmetric processes using shared variables. *Distributed Computing*, 8(1):19–38, 1994.
16. D. Goldschlag, M. Reed, and P. Syverson. Onion routing. *Commun. ACM*, 42(2):39–41, 1999.
17. R. Guerraoui and E. Ruppert. What can be implmented anonymously? Technical Report 200496, School of Computer and Communications Sciences, EPFL, 2004.
18. M. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, 1991.
19. M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd IEEE Intl Conf. on Distributed Computing Systems*, pages 522–529, 2003.
20. M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, 1999.
21. M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.
22. P. Jayanti and S. Toueg. Wakeup under read/write atomicity. In *Distributed Algorithms, 4th Intl Workshop*, volume 486 of *LNCS*, pages 277–288, 1990.
23. R. E. Johnson and F. B. Schneider. Symmetry and similarity in distributed systems. In *4th ACM Symp. on PODC*, pages 13–22, 1985.
24. S. Kutten, R. Ostrovsky, and B. Patt-Shamir. The Las-Vegas processor identity problem (How and when to be unique). *J. Algs*, 37(2):468–494, 2000.
25. R. J. Lipton and A. Park. The processor identity problem. *Inf. Process. Lett.*, 36(2):91–94, 1990.
26. M. C. Loui and H. H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In F. P. Preparata, editor, *Advances in Computing Research*, volume 4, pages 163–183. JAI Press, Greenwich, Connecticut, 1987.
27. G. Neiger. Set-linearizability. In *13th ACM Symp. on PODC*, page 396, 1994.
28. A. Panconesi, M. Papatriantafilou, P. Tsigas, and P. Vitányi. Randomized naming using wait-free shared variables. *Distributed Computing*, 11(3):113–124, 1998.
29. M. K. Reiter and A. D. Rubin. Crowds: Anonymity for web transactions. *ACM Trans. on Inf. and System Security*, 1(1):66–92, 1998.
30. S.-H. Teng. Space efficient processor identity protocol. *Inf. Process. Lett.*, 34(3):147–154, 1990.