What Can Be Implemented Anonymously?

Rachid Guerraoui
EPFL
Lausanne, Switzerland

Eric Ruppert
York University
Toronto, Canada

November 12, 2004

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? Here, we consider asynchronous shared-memory systems that are anonymous, which means processes do not have identifiers, and are programmed identically. The shared memory contains only the most common type of shared objects, read/write registers. We investigate what can be computed deterministically in this model when any number of processes can experience crash failures. Thus, an adversary controls the speeds of processes and the failures in the system. Typically algorithms designed for such systems guarantee (partial) correctness in all possible executions, and there are various types of termination (or progress) properties that have been studied: wait-freedom, lock-freedom and obstruction-freedom. We give anonymous algorithms for some of the most important problems in the theory of distributed computing: timestamping, atomic snapshots and consensus. Our solutions to the first two are wait-free and the third is obstruction-free. We also show that a shared data structure has an obstruction-free implementation in our model 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 data structure.

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), on their means of communication (e.g., shared memory or message passing), and on their failure modes (e.g., crash failures or malicious failures) have

led to an abundant literature. Virtually all of this literature assumes that processes have distinct identities. Besides intellectual curiosity, it is practically appealing to revisit this fundamental assumption. Indeed, certain distributed applications, like web servers [26] and peer-to-peer file sharing systems [10], do sometimes mandate preserving the anonymity of the users and forbid the use of any form of identity in the distributed computation. (See [9] for a discussion of anonymous computing used for privacy.) Other systems, like sensor networks, consider mass-produced tiny agents that are not even equipped with identifiers [4].

In an *anonymous* system, processes do not have identifiers and are programmed identically. There has been work on anonymous message-passing systems, starting with Angluin [3], but very little research has looked at anonymous shared-memory systems. The work that does exist on shared-memory systems has assumed failure-free systems or the existence of a random oracle to build randomized algorithms. (See Section 2 for a discussion of this work.)

The goal of this paper is to determine what types of shared data structures (called objects) can be implemented *deterministically* in an anonymous shared-memory system where failures can occur. We consider the model where processes communicate with one another using the most basic type of shared-memory objects: read/write registers. We assume that any number of processes may experience unpredictable crash failures. 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 *are* solvable without having to break symmetry. Since we are interested in totally anonymous systems, we assume that registers are multi-writer, so that every process is permitted to write to every register. (In contrast, usage of single-writer registers would give processes at least some rudimentary sense of identity. For example, every process would know that values written into the same register at different times were produced by the same process.)

There are several properties that can be used to guarantee that a system will make progress regardless of failures or asynchrony. The strongest is *wait-freedom* [15], which requires *every* non-faulty process to complete its algorithm in a finite number of its own steps. However, there are many cases where wait-free algorithms are provably impossible or are too inefficient to be practical. Often, a weaker progress guarantee is sufficient. *Lock-freedom* is one such condition: it guarantees 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 than lock-freedom is *obstruction-freedom* [16], which can be very useful when low contention is expected to be the common case, or if some contention-management system is used. Obstruction-freedom guarantees that a process will complete its algorithm whenever it has an opportunity to take enough steps without interruption from other processes.

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. Most other

objects have no wait-free (or even lock-free) implementation [15]. 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 have an obstruction-free implementation [16]. We show that the anonymous 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.

Many distributed algorithms make use of timestamps to help processes agree on the order of various events in the system. Objects, such as fetch&increment objects 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 some interesting applications. We construct, in Section 4, an efficient, wait-free implementation of a weak counter from registers.

In Section 5, we consider the problem of implementing an atomic snapshot object [1, 2, 6], which is an abstraction of the problem of obtaining a consistent view of many registers while they are being updated by other processes. It is used as a fundamental building block of many other algorithms. Many implementations of snapshot objects have been published in the literature and, to our knowledge, all of them do make essential use of process identities. In fact, at first glance, it was not clear whether a wait-free atomic snapshot implementation was possible in anonymous systems. Wait-free algorithms generally make use of helping mechanisms, in which fast processes help the slow ones to 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. We show that a wait-free snapshot implementation does exist and is reasonably efficient in terms of time complexity. The timestamps provided by the weak counter object are essential in this construction. We also give a lock-free implementation with better space complexity.

In Section 6, we consider the consensus problem, which requires processes with private input values to agree on one of those values. It is embedded in a wide variety of process-coordination tasks. There is no wait-free implementation of consensus using shared registers, even if processes do have identifiers [15, 23]. We sketch an obstruction-free anonymous consensus algorithm that is essentially a derandomized version of the randomized wait-free anonymous consensus algorithm of Buhrman *et al.* [11]. This algorithm uses unbounded space. We construct an algorithm that uses bounded space, with the help of our snapshot implementation.

Finally, we give a complete characterization of the types of objects that have obstruction-free implementations in our model in Section 7. An object can be implemented if and only if 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 that this condition is necessary. To prove sufficiency, we give a "universal" construction that implements any

object satisfying the condition. This construction makes use of our weak counter object, as well as our consensus algorithm.

2 Related Work

There has been research that studies anonymous shared-memory systems when no failures can occur. Johnson and Schneider [20] gave some leader election algorithms using versions of single-writer snapshots and test&set objects. Attiya, Gorbach and Moran [8] studied anonymous systems that are equipped with registers and have no failures. They gave a characterization of the tasks that are solvable in this model if n is not known. The characterization is the same if n is known [13]. Consensus is solvable in these models, but it is not solvable if the registers cannot be initialized by the programmer [19]. Aspnes, Fich and Ruppert [5] looked at failure-free models that contain other types of shared objects, such as counters and also characterized which shared-memory models can be implemented if communication is through anonymous broadcasts. They showed that this broadcast model is equivalent to a shared-memory system with counters and strictly stronger than a shared-register model.

There has also been some research on randomized algorithms for anonymous shared-register systems with no failures. For the *naming* problem, processes must choose unique names for themselves. There is a simple naming algorithm in which processes randomly choose the names so that they will be unique with high probability. Shared registers can be used to detect when the names chosen are indeed unique, thus guaranteeing unique names whenever the algorithm terminates, which happens with high probability [22, 27]. Two papers gave randomized renaming algorithms that have finite expected running time, and hence terminate with probability 1 [14, 21].

Randomized algorithms for systems with halting failures have also been studied. Panconesi *et al.* [25] 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 [11, 14, 21]. Interestingly, Buhrman *et al.* [11] gave a randomized wait-free algorithm for consensus in this model that is based on Chandra's randomized consensus algorithm [12]. Thus, even consensus does not help to give processes unique identifiers. Aspnes, Shah and Shah [7] extended the algorithm of Buhrman *et al.* to a setting where there are infinitely many anonymous 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 [17] gave a characterization of the decision tasks 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 [15]. 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 sufficiency proof in Section 7.

3 Model

We consider an *anonymous* system, where a collection of n processes execute identical algorithms, and 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*.

The 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. It is assumed that the programmer can choose the initial state of the objects used. With the exception of the weak counter object we introduce in Section 4, all shared objects we consider are linearizable [18]: 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 can atomically change the state of an object and return a response to the process that invoked the operation. (The weak counter object can be viewed as a set-linearizable object [24].)

Some types of objects are provided by the system and all other types of objects needed must be implemented from the base types. 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 must execute identical code to perform a particular operation. In addition, the implementation must 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.

We assume the shared memory contains the most basic kind of shared 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 of the object to v. 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, we use the convention that the names of shared

objects begin with an upper-case letter, while lower-case letters are used for the process's private variables.

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 Section 5 and our characterization of implementable types in Section 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 [15]. By considering our weaker version, we have an object that is implementable, and still strong enough for our purposes.

We show that there is 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 progress property of lock-freedom, but uses only binary registers, is then described briefly. Processes must know the value of n (or at least an upper bound on it) for the wait-free implementation, but this knowledge is unnecessary for the lock-free algorithm.

Our wait-free implementation uses an array $A[1, 2, \ldots]$ of binary registers, each initialized to $\perp$. 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 the pseudocode in Figure 1.) The key property for correctness is an invariant that is maintained at all times: if $A[k] = \top$, then all entries in $A[1..k]$ are $\top$. To locate the first $\perp$ in A efficiently, the algorithm uses a binary search. Starting from the location a found by the process's previous GetTimestamp operation (or from location 1 if there was no such operation), the algorithm probes locations $a + 1, a + 3, a + 7, \ldots, a + 2^i - 1, \ldots$ until it finds a $\perp$ in location b. 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 avoid the possibility that a process can 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 ever sees

GetTimestamp

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

Figure 1: Wait-free implementation of a weak counter from registers.

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. Each process's local variable a should be initialized to 1.

Theorem 1 *Figure 1 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 a $\top$ is written into $A[b]$ by line 17, we have $a = b$,

so the entry $A[b-1]$ 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*.

It is easy to prove Invariant 3 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.

Proposition 2 *If n processes perform a total of k invocations of the* GETTIMESTAMP *algorithm in Figure 1, the worst-case total number of steps by all processes is $\Theta(k \log n)$.*

Proof: We use an amortized analysis. To do this analysis, we shall count only the number of locations probed in line 3, 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 ith 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 $2^{-i+2} < 4 \cdot 2^{-\log n} = \frac{4}{n}$.

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 6 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 $4 + \log n$.

To see that this analysis is tight (when $k > n$), consider an execution where the following sequence is repeated $\left\lfloor \frac{k}{n} \right\rfloor$ 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 $\Omega(\frac{k}{n}(n + (n-1)\log n)) = \Omega(k \log n)$ steps in total. ■

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 lock-free 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 lock-free, 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 $O(k \log n)$.*

5 Snapshot Objects

The snapshot object was introduced independently by Afek *et al.* [1], Anderson [2], and Aspnes and Herlihy [6]. It has proved to be 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 is a celebrated example of an object type that has wait-free implementations from registers, and has been widely used as the 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 extensively studied, and many

algorithms exist to implement them, but all require processes to have unique identifiers. Our first construction is a fairly simple modification of the standard lock-free snapshot algorithm for non-anonymous systems [1].

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

Proof: One register is used to represent each component of the snapshot object. Each process keeps a local variable t that stores a timestamp. To update 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 $(v_1, v_2, \ldots, v_m)$.

Lock-freedom: UPDATES terminate in a single step. A SCAN can only be prevented from terminating if UPDATES are constantly being completed.

Correctness: 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 ith 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 the value of the jth register is different from (t_j, v_j) at 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 jth register contains (t_j, v_j). Linearize the SCAN at that moment. ∎

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. Here, 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) shared multi-writer 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 Figure 2.

Theorem 5 *The algorithm in Figure 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: **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 $n-1$

```
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 + 1 sets of reads returned identical results
7            then return the first field of each value
8          end if
9  end loop
```

Figure 2: Wait-free implementation of a snapshot object from registers.

changes to the registers $R_1, \ldots, R_m$ can be observed before the first termination condition is satisfied. If the first condition is never satisfied, then after at most $n^2 + 1$ iterations of the loop, the SCAN will see the same values in $n+1$ sets of reads, so the loop can terminate using the second termination condition. Each operation terminates within $O(mn^2)$ steps, plus the time required for the GETTIMESTAMP operation. It follows from Theorem 2 that the total number of steps in an execution with k invocations of UPDATES and SCANS is $O(k \log n + kmn^2) = O(kmn^2)$.

Correctness: 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 $n + 1$ sets of reads. None of the triples r_i will contain a timestamp greater than t (otherwise P would have terminated after the first set). Suppose the first of the n sets of reads begins at time T. Then 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 between two of the identical sets of reads, no UPDATE performed a write, so between those two sets of reads, the value of R_i was r_i, for all i. If we linearize the SCAN between those two sets of reads, 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 Obstruction-Free 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 referred to as *agreement* and *validity*, respectively.

Herlihy, Luchangco and Moir [16] observed that a randomized algorithm to solve wait-free consensus can be "derandomized" to obtain an obstruction-free consensus algorithm. If we derandomize the *anonymous* consensus algorithm of Buhrman *et al.* [11], we obtain the following theorem.

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

Proof: We begin with an algorithm for binary consensus and then use it to build an algorithm for multi-valued consensus. The binary consensus algorithm, given in Figure 3, is similar to the randomized wait-free algorithm of Buhrman *et al.* [11]. It 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 jth iteration of the loop as round j. A process is said to *change its preference* whenever it executes line 8.

We first prove the following invariant: For $v \in \{0, 1\}, j \geq 1$, $R_v[j+1] \neq \perp \Rightarrow R_v[j] \neq \perp$. 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.

Obstruction-freedom: Suppose some process P begins running by itself from some configuration C. Let $\hat{\jmath}$ be some value such that $R_0[\hat{\jmath}] = R_1[\hat{\jmath}] = \perp$ in C. Eventually, P's value of j will increase to $\hat{\jmath}$. It will write into either $R_0[\hat{\jmath}]$ or $R_1[\hat{\jmath}]$ and then decide in its next round.

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

Figure 3: Obstruction-free consensus binary consensus using binary registers.

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 could not have changed 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 $\hat{\jmath}$ be the smallest round in which some process decides. Let P be any process that decides in round $\hat{\jmath}$. 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 $R_1[\hat{\jmath} - 1] = \perp$. This means that no process could have changed its preference from 0 to 1 during round $\hat{\jmath} - 1$ before time T.

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

Thus, all processes that enter round $\hat{\jmath}$ with preference 1 do so after T, so they will all change their preferences to 0 during round $\hat{\jmath}$, and no process can ever write to $R_1[\hat{\jmath}]$ during the entire execution. It follows that each process that enters round $\hat{\jmath}$ 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 $\hat{\jmath} + 1$ do so with preference 0, and they will all decide 0 during that round. ■

The construction of Theorem 6 requires an unbounded number of binary registers. In this section, we give an obstruction-free algorithm for anonymous 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 simply 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 can stop and output 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 $4n + 1$ 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.

There are several complications 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 Figure 4, where we use $\bar{v}$ to denote $1 - v$.

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

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

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

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

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. (It is not hard to see that 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 4 will always fail, so (2) can never become false, and this means that no process can every write to R_1 in line 15. Thus any process that decides in line 8 can only decide 0.

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, after T, there is always a value bigger than or equal to m in R_0, and a value smaller than m in R_1. Thus, $\min_{1 \leq i \leq 4n+1} R_1[i] < m \leq \max_{1 \leq i \leq 4n+1} R_0[i]$ at all times after T. So 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 the case of non-binary consensus using the following proposition. The proposition is proved using a common technique of agreeing 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 $\log |D|$ copies of S and $|D|$ additional binary registers. It can also be done using $\log |D|$ copies of S and $2 \log |D|$ additional registers 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 bits of the output, $b_1, \ldots, b_m$. Assume also that every process stores (locally) a preference that is the input of some process and begins with $b_1 b_2 \ldots b_m$. Processes execute binary consensus, using the $(m+1)$th bit of their preferred value as the input, to decide on the next bit of the output, b_{m+1}. If the value agreed upon is different from the value a process proposed, that process must update its preference to one that agrees with the first $m+1$ bits that have been decided. It can do this by searching the array R.

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 $R_0[1..\log |D|]$ and $R_1[1..\log |D|]$. Before a process proposes a value b to the $(m+1)$th instance of binary consensus, which is being used to determine the $(m+1)$th bit of the output, it writes its preference into $R_b[m]$. If a process must change its preference as a result of the binary consensus, it uses the value it then reads from $R_{\bar{b}}[m]$. ■

Corollary 9 *There is an anonymous, obstruction-free algorithm for consensus, with arbitrary inputs, using binary 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].)

We now make this definition of idempotence 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.

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: ($\Rightarrow$) Assume there is an anonymous, obstruction-free implementation of T from binary registers. 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, followed by op. (This execution must terminate, since the implementation is obstruction-free.) Since the object is deterministic, P 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 P and Q execute the code for op, taking alternate steps. Since P and Q 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 used by the implementation 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. This proves that T is idempotent.

($\Leftarrow$) 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 Figure 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
```

Figure 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.

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 given above.

We must still show that the linearization point chosen for a simulated operation is within the interval of time when the operation is running. Let D be an execution of DO(op) which belongs to group i. The linearization point T chosen for 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.*

It is interesting to note that, in the more often-studied context of non-anonymous wait-free computing, counters can be implemented from registers [6], while consensus objects cannot [15, 23]. 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 [15] will not tell us very much about anonymous, obstruction-free implementations since, for example, consensus objects cannot implement counters, which have consensus number 1.

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. Part of this work was done while Eric Ruppert was visiting EPFL.

References

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

[2] James H. Anderson. Composite registers. *Distributed Computing*, 6(3), pages 141–154, April 1993.

[3] Dana Angluin. Local and global properties in networks of processors. In *Proceedings of the 12th ACM Symposium on Theory of Computing*, pages 82–93, 1980.

[4] Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. In *Proceedings of 23rd ACM Symposium on Principles of Distributed Computing*, pages 290–299, 2004.

[5] James Aspnes, Faith Fich, and Eric Ruppert. Relationships between broadcast and shared memory in reliable anonymous distributed systems. In *Distributed Computing, 18th International Symposium*, pages 260–274, 2004.

[6] James Aspnes and Maurice Herlihy. Wait-free data structures in the asynchronous PRAM model. In *Proc. 2nd ACM Symposium on Parallel Algorithms and Architectures*, pages 340–349, 1990.

[7] James Aspnes, Gauri Shah, and Jatin Shah. Wait-free consensus with infinite arrivals. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 524–533, 2002.

[8] Hagit Attiya, Alla Gorbach, and Shlomo Moran. Computing in totally anonymous asynchronous shared memory systems. *Information and Computation*, 173(2), pages 162–183, March 2002.

[9] Oliver Berthold, Hannes Federrath, and Marit Köhntopp. Project "anonymity and unobservability in the internet". In *Proc. 10th Conference on Computers, Freedom and Privacy*, pages 57–65, 2000.

[10] Stephen C. Bono, Christopher A. Soghoian, and Fabian 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.

[11] Harry Buhrman, Alessandro Panconesi, Riccardo Silvestri, and Paul Vitanyi. On the importance of having an identity or, is consensus really universal? In *Distributed Computing, 14th International Conference*, volume 1914 of *LNCS*, pages 134–148, 2000.

[12] Tushar Deepak Chandra. Polylog randomized wait-free consensus. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pages 166–175, 1996.

[13] Catalin Drulă. The totally anonymous shared memory model in which the number of proces ses is known. Personal communication.

[14] Ömer Eğecioğlu and Ambuj K. Singh. Naming symmetric processes using shared variables. *Distributed Computing*, 8(1), pages 19–38, 1994.

[15] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1), pages 124–149, January 1991.

[16] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems*, pages 522–529, 2003.

[17] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6), pages 858–923, November 1999.

[18] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3), pages 463–492, July 1990.

[19] Prasad Jayanti and Sam Toueg. Wakeup under read/write atomicity. In *Distributed Algorithms, 4th International Workshop*, volume 486 of *LNCS*, pages 277–288, 1990.

[20] Ralph E. Johnson and Fred B. Schneider. Symmetry and similarity in distributed systems. In *Proc. 4th ACM Symposium on Principles of Distributed Computing*, pages 13–22, 1985.

[21] Shay Kutten, Rafail Ostrovsky, and Boaz Patt-Shamir. The Las-Vegas processor identity problem (How and when to be unique). *Journal of Algorithms*, 37(2), pages 468–494, November 2000.

[22] Richard J. Lipton and Arvin Park. The processor identity problem. *Information Processing Letters*, 36(2), pages 91–94, October 1990.

[23] Michael C. Loui and Hosame H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In Franco P. Preparata, ed., *Advances in Computing Research*, volume 4, pages 163–183. JAI Press, Greenwich, Connecticut, 1987.

[24] Gil Neiger. Set-linearizability. In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing*, page 396, 1994.

[25] Alessandro Panconesi, Marina Papatriantafilou, Philippas Tsigas, and Paul Vitányi. Randomized naming using wait-free shared variables. *Distributed Computing*, 11(3), pages 113–124, August 1998.

[26] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security*, 1(1), pages 66–92, November 1998.

[27] Shang-Hua Teng. Space efficient processor identity protocol. *Information Processing Letters*, 34(3), pages 147–154, April 1990.