The Space Complexity of Unbounded Timestamps

Faith Ellen[1], Panagiota Fatourou[2], and Eric Ruppert[3]

[1] University of Toronto, Canada
[2] University of Ioannina, Greece
[3] York University, Canada

Abstract. The timestamp problem captures a fundamental aspect of asynchronous distributed computing. It allows processes to label events throughout the system with timestamps that provide information about the real-time ordering of those events. We consider the space complexity of wait-free implementations of timestamps from shared read-write registers in a system of n processes.
We prove an $\Omega(\sqrt{n})$ lower bound on the number of registers required. If the timestamps are elements of a nowhere dense set, for example the integers, we prove a stronger, and tight, lower bound of n. However, if timestamps are not from a nowhere dense set, this bound can be beaten; we give an algorithm that uses $n-1$ (single-writer) registers.
We also consider the special case of anonymous algorithms, where processes do not have unique identifiers. We prove anonymous timestamp algorithms require n registers. We give an algorithm to prove that this lower bound is tight. This is the first anonymous algorithm that uses a finite number of registers. Although this algorithm is wait-free, its step complexity is not bounded. We also present an algorithm that uses $O(n^2)$ registers and has bounded step complexity.
Keywords: timestamps, shared memory, anonymous, lower bounds.

1 Introduction

In asynchronous systems, it is the unpredictability of the scheduler that gives rise to the principle challenges of designing distributed algorithms. One approach to overcoming these challenges is for processes to determine the temporal ordering of certain events that take place at different locations within the system. Examples of tasks where such temporal information is essential include implementing first-come first-served processing of jobs that arrive at different locations in the system and knowing whether a locally cached copy of data is up-to-date. Temporal information about the scheduling of events can also be used to break symmetry, *e.g.*, the first process to perform some step can be elected as a leader.

If processes communicate via messages or shared read-write registers, it is impossible for them to determine the exact temporal ordering of all events. However, timestamps provide partial information about this ordering in such systems. A timestamp algorithm allows processes to ask for labels, or *timestamps*, which can then be compared with other timestamps. Timestamps have been used to solve several of the most fundamental problems in distributed computing. Examples include mutual exclusion [17] (and the more general k-exclusion problem [2]), randomized consensus [1], and constructing multi-writer registers from single-writer registers [13, 19, 22]. Timestamps have also been employed in anonymous systems as building blocks for implementations of wait-free atomic snapshots and other data structures [12].

Despite the central importance of the timestamp problem, its complexity is not well-understood. In this paper, we present the first study on the number of registers required for wait-free implementations of timestamps.

The history of timestamps begins with Lamport [18], who defined a partial ordering on events in a message-passing system; one event "happens before" another if the first could influence the second (because they are by the same process or because of messages sent between processes). He defined a logical clock, which assigns integer timestamps to events such that, if one event happens before another, it is assigned a smaller timestamp. There is no constraint on the relationship between timestamps assigned to other pairs of events.

Fidge and Mattern [11, 20] introduced the notion of vector clocks, where timestamps are vectors of integers rather than integers. Two vectors are compared component-wise: one vector is smaller than or equal to another when each component of the first is smaller than or equal to the corresponding component of the second. Their vector clock algorithms satisfy the property that one event gets a smaller vector than another if *and only if* it happens before the other event. This property is not possible to ensure using integer timestamps, because concurrent events may need to be assigned incomparable vectors. Charron-Bost [5] proved that the number of components required by a vector clock is at least the number of processes, n.

In message-passing algorithms, the timestamps reflect the partial order representing (potential) causal relationships. In shared-memory systems, we are concerned, instead, with the real-time ordering of events.

The simplest shared-memory timestamp algorithm uses single-writer registers [17]. To get a new timestamp, a process collects the values in all the single-writer registers and writes one plus the maximum value it read into its single-writer register. This value is its new timestamp.

Dwork and Waarts [8] described a vector timestamp algorithm that uses n single-writer registers. To obtain a new timestamp, a process increments its register and collects the values in the registers of all other processes. It returns the vector of these n values as its timestamp. These timestamps can be compared either lexicographically or in the same way as in the vector clock algorithm.

Attiya and Fouren [3] gave a vector timestamp algorithm that is considerably more complicated. It uses an unbounded number of registers but has the advantage that the number of components in the timestamp (and the time required to obtain it) is a function of the number of processes running concurrently.

Guerraoui and Ruppert [12] described an anonymous wait-free timestamp algorithm, but the number of registers used and the time-complexity of getting a timestamp increases without bound as the number of labelled events increases. Thus, their algorithm is not bounded wait-free.

In all the above algorithms, the size of timestamps grows without bound as the number of labelled events increases. This is necessary to describe the ordering among an unbounded number of non-concurrent events. For some applications, one can restrict the events about which order queries can be made, for example, only the most recent event by each process. This restriction allows timestamps to be reused, so they can be of bounded size. This restricted version of timestamps is called the *bounded timestamp* problem. In contrast, the general version of the problem is sometimes called the *unbounded timestamp* problem. Israeli and Li [14] gave a bounded timestamp algorithm, assuming timestamps are only generated by one process at a time. Dolev and Shavit defined and solved the problem allowing multiple processes to obtain timestamps concurrently [6]. This and other known implementations of bounded concurrent timestamps [7, 8, 13, 15] are quite complex, as compared to unbounded timestamps.

It is known that bounded timestamp algorithms must use $\Omega(n)$ bits per timestamp [14]. In contrast, unbounded timestamp algorithms can use timestamps whose bit lengths are logarithmic in the number of events that must be labelled. Thus, if the number of events requiring timestamps is reasonable (for example, less than 2^{64}), timestamps will easily fit into one word of memory. The work on the bounded timestamp problem is of great interest and technical depth. However, since bounded timestamp algorithms are complicated and require long timestamps, the unbounded version is often considered more practical. This paper focusses exclusively on the unbounded timestamp problem.

1.1 Our Contributions

In this paper, we study the number of read-write registers needed to implement timestamps. We present both upper and lower bounds. For our upper bounds, we give wait-free algorithms. The lower bounds apply even if algorithms must only satisfy the weaker progress property of obstruction-freedom. Our most general lower bound shows that any timestamp algorithm must use $\Omega(\sqrt{n})$ registers. Previously known wait-free algorithms use n registers. We show how to modify one of these algorithms to use $n-1$ registers.

Some existing timestamp implementations use timestamps drawn from a nowhere dense set. Intuitively, this means that between any two possible timestamps, there are a finite number of other timestamps. For this restricted case, we show that any such implementation must use at least n registers, exactly matching known implementations. Interestingly, our lower bound can be beaten by using timestamps from a domain that is not nowhere dense, namely, pairs of integers, ordered lexicographically.

We also prove matching upper and lower bounds for anonymous systems, where processes do not have unique identifiers and are programmed identically. We give a wait-free algorithm using n registers, whereas previous algorithms used an unbounded number. We also provide another, faster anonymous algorithm. It uses $O(n^2)$ registers and a process takes $O(n^3)$ steps to obtain a timestamp.

We prove a tight lower bound of n for the number of registers required for an anonymous timestamp implementation. This establishes a small but interesting space complexity separation between the anonymous and general versions of the timestamp problem, since $n-1$ registers suffice for our algorithm, which uses identifiers. Lower bounds for anonymous systems are interesting, in part, because they provide insight for lower bounds in more general systems [9, 10].

Guerraoui and Ruppert [12] used timestamps as a subroutine for their anonymous implementation of a snapshot object. Plugging in our space-optimal anonymous timestamp algorithm yields an anonymous wait-free implementation of an m-component snapshot from $m+n$ registers. This is the first such algorithm to use a bounded number of registers. Similarly, if our second anonymous timestamp algorithm is used, we obtain an anonymous wait-free snapshot implementation from $O(m+n^2)$ registers where each Scan and Update takes $O(n^2(m+n))$ steps. This is the first bounded wait-free anonymous snapshot implementation.

2 The Model of Computation

We use a standard model for asynchronous shared-memory systems, in which a collection of n processes communicate using atomic read-write registers. We consider only *deterministic* algorithms. If processes have identical programmes and do not have unique identifiers, the algorithm is called *anonymous*; otherwise, it is called *eponymous* [21]. An *execution* of an algorithm is a possibly infinite sequence of steps, where each step is an access to a shared register by some process, followed by local computation of that process. The subsequence of steps taken by each process must conform to the algorithm of that process. Each read of a register returns the value that was most recently written there (or the initial value of the register if no write to it has occurred). If $\mathcal{P}$ is a set of processes, a *$\mathcal{P}$-only* execution is an execution in which only processes in $\mathcal{P}$ take steps. A *solo execution* by a process p is a $\{p\}$-only execution. We use $\alpha \cdot \beta$ to denote the concatenation of the finite execution α and the (finite or infinite) execution β. A *configuration* is a complete description of the system at some time. It is comprised of the internal state of each process and the value stored in each shared register. A configuration C is *reachable* if there is an execution from an initial configuration that ends in C. In an execution, two operation instances are called *concurrent* if neither one ends before the other begins.

We consider processes that may fail by halting. An algorithm is *wait-free* if every non-faulty process completes its tasks within a finite number of its own steps, no matter how processes are scheduled or which other processes fail. A stronger version of the wait-freedom property, called *bounded wait-freedom*, requires that the number of steps be bounded. A much weaker progress property is *obstruction-freedom*, which requires that each process must complete its task if it is given sufficiently many consecutive steps.

In our algorithms, each register need only be large enough to store one timestamp. For our lower bounds, we assume that each register can hold arbitrarily large amounts of information. In our algorithms, we use the convention that shared registers have names that begin with upper-case letters and local variables begin with lower-case letters. If $\mathcal{R}$ is a set or array of registers, we use $\textsc{Collect}(\mathcal{R})$ to denote a read of each register in $\mathcal{R}$, in some unspecified order.

Our lower bounds use covering arguments, introduced by Burns and Lynch [4]. We say a process p *covers* a register R in a configuration C if p will write to R when it next takes a step. A set of processes $\mathcal{P}$ *covers* a set of registers $\mathcal{R}$ in C if $|\mathcal{P}| = |\mathcal{R}|$ and each register in $\mathcal{R}$ is covered by exactly one process in $\mathcal{P}$. If $\mathcal{P}$ covers $\mathcal{R}$, a *block write* by $\mathcal{P}$ is an execution in which each process in $\mathcal{P}$ takes exactly one step writing its value.

3 The Timestamp Problem

A *timestamp implementation* provides two algorithms for each process: GetTS and Compare. GetTS takes no arguments and outputs a value from a universe U. Elements of U are called *timestamps*. Compare takes two arguments from U and outputs a Boolean value. If an instance of GetTS, which outputs t_1, finishes before another instance, which outputs t_2, begins, then any subsequent instances of $\textsc{Compare}(t_1, t_2)$ and $\textsc{Compare}(t_2, t_1)$ must output true and false, respectively. Thus, two non-concurrent GetTS operations cannot return the same timestamp. Unlike the bounded timestamp problem, Compare can compare any previously granted timestamps, so U must be infinite.

This definition of the timestamp problem is weak, which makes our lower bounds stronger. It is sufficient for some applications [12], but it is too weak for other applications. For example, consider the implementation of atomic multi-writer registers from single-writer registers [13, 19, 22]. Suppose readers determine which value to return by comparing timestamps attached to each written value to find the most recently written value. If two writers write different values concurrently, and two readers later read the register, the readers should agree on which of the two values to return. To handle this kind of application, we can define a stronger version of the timestamp problem which requires that, for each pair t and t', all $\textsc{Compare}(t, t')$ operations in the same execution must return the same value. A *static* timestamp algorithm is one that satisfies a still stronger property: for each pair, t and t', the $\textsc{Compare}(t, t')$ always returns the same result in *all* executions. Static timestamp algorithms have the nice property that Compare queries need not access shared memory. The algorithms we present in this paper are all static. The lower bounds in Sections 4.1 and 7 apply even for non-static implementations.

A natural way to design a static timestamp algorithm is to use timestamps drawn from a partially ordered universe U, and answer Compare queries using that order; $\textsc{Compare}(t_1, t_2)$ returns true if and only if $t_1 < t_2$. A partially ordered set U is called *nowhere dense* if, for every $x, y \in U$, there are only a finite number of elements $z \in U$ such that $x < z < y$. The integers, in their natural order, and the set of all finite sets of integers, ordered by set inclusion, are nowhere dense. Any set of fixed-length vectors of integers, where $x \leq y$ if and only if each component of x is less than or equal to the corresponding component of y is too. However, for $k \geq 2$, the set of all length-k vectors of integers, ordered lexicographically, is not nowhere dense.

Another desirable property is that all timestamps produced are distinct, even for concurrent GetTS operations. In eponymous systems, this property is easy to satisfy by incorporating the process's identifier into the timestamp generated [17]. In anonymous systems, it is impossible, because symmetry cannot be broken using registers.

4 Eponymous Lower Bounds

We prove lower bounds on the number of registers needed to implement timestamps eponymously. First, we give the most general result of the paper, proving that $\Omega(\sqrt{n})$ registers are needed. Then, we prove a tight lower bound of n if the timestamps are chosen from a partially ordered set that is nowhere dense.

4.1 A General Space Lower Bound

We use a covering argument, showing that, starting from a configuration where some registers are covered, we can reach another configuration where more registers are covered. The following lemma allows us to do this, provided the original registers are covered by *three* processes each. The complement of a set of processes $\mathcal{S}$ is denoted by $\overline{\mathcal{S}}$.

Lemma 1. *Consider any timestamp algorithm. Suppose that, in a reachable configuration C, there are three disjoint sets of processes, $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{Q}$ that each cover the set of registers $\mathcal{R}$. Let C_i be the configuration obtained from C by having the processes in $\mathcal{P}_i$ do a block write, β_i, for $i = 1, 2$. Then for all disjoint sets $\mathcal{S}_1 \subseteq \overline{\mathcal{P}_2 \cup \mathcal{Q}}$ and $\mathcal{S}_2 \subseteq \overline{\mathcal{P}_1 \cup \mathcal{Q}}$, with some process not in $\mathcal{S}_1 \cup \mathcal{S}_2$, there is an $i \in \{1, 2\}$ such that every $\mathcal{S}_i$-only execution starting from C_i that contains a complete* GetTS *writes to a register not in $\mathcal{R}$.*

Proof. Suppose there exist disjoint sets $\mathcal{S}_1 \subseteq \overline{\mathcal{P}_2 \cup \mathcal{Q}}$ and $\mathcal{S}_2 \subseteq \overline{\mathcal{P}_1 \cup \mathcal{Q}}$, an $\mathcal{S}_1$-only execution α_1 from C_1 and an $\mathcal{S}_2$-only execution α_2 from C_2 that both write only to registers in $\mathcal{R}$, and $q \notin \mathcal{S}_1 \cup \mathcal{S}_2$. Also suppose α_1 and α_2 contain complete instances of GetTS, I_1 and I_2, that return t_1 and t_2, respectively. Let γ be an execution starting from C that begins with a block write to $\mathcal{R}$ by $\mathcal{Q}$, followed by a solo execution in which q performs a complete instance of Compare(t_1, t_2). Then, $\beta_1 \cdot \alpha_1 \cdot \beta_2 \cdot \alpha_2 \cdot \gamma$ and $\beta_2 \cdot \alpha_2 \cdot \beta_1 \cdot \alpha_1 \cdot \gamma$ are valid executions starting from C that are indistinguishable to q. Hence, in both, q returns the same result for Compare(t_1, t_2). This is incorrect, since I_1 precedes I_2 in $\beta_1 \cdot \alpha_1 \cdot \beta_2 \cdot \alpha_2 \cdot \gamma$, but I_2 precedes I_1 in $\beta_2 \cdot \alpha_2 \cdot \beta_1 \cdot \alpha_1 \cdot \gamma$. □

Theorem 2. *Every obstruction-free timestamp algorithm for n processes uses more than $\frac{1}{2}\sqrt{n-1}$ registers.*

Proof. First, we show that at least one register is required. To derive a contradiction, suppose there is an implementation that uses no shared registers. Let α and β be solo executions of GETTS by different processes, p and q, starting from the initial configuration. Suppose they return timestamps t and t'. Let γ be a solo execution of COMPARE(t, t') by p immediately following α. Since $\alpha \cdot \beta \cdot \gamma$ is indistinguishable from $\beta \cdot \alpha \cdot \gamma$ to p, it must return the same result for COMPARE(t, t') in both. However, it must return true in $\alpha \cdot \beta \cdot \gamma$ and false in $\beta \cdot \alpha \cdot \gamma$. This is a contradiction. This suffices to prove the claim for $n \leq 4$.

For the remainder of the proof, we assume that $n \geq 5$. Consider any timestamp algorithm that uses $r > 0$ registers. To derive a contradiction, assume $r \leq \frac{1}{2}\sqrt{n-1}$. We show, by repeated applications of Lemma 1 that it is possible to reach a configuration where all r registers are covered by three processes each. One further application of Lemma 1 will then show that some process must write to some other register, to produce the desired contradiction. We prove the following claim by induction on k.

Claim: For $k = 1, \ldots, r$, there is a reachable configuration with k registers each covered by $r - k + 3$ processes.

Base case ($k = 1$): Let p_1, p_2 and q be any three processes. Applying Lemma 1 with initial configuration C, $\mathcal{R} = \emptyset$, $\mathcal{P}_1 = \mathcal{P}_2 = \mathcal{Q} = \emptyset$, $\mathcal{S}_1 = \{p_1\}$, and $\mathcal{S}_2 = \{p_2\}$ proves that the solo execution of GETTS by either p_1 or p_2 must write to some register. Thus, all except possibly one process must write to a register during a solo execution of GETTS starting from C. Consider an execution consisting of the concatenation of the longest write-free prefixes of $n - 1$ of these solo executions. In the resulting configuration, there are $n - 1$ processes covering registers. Since there are r registers and $n - 1 \geq (2r)^2 > r(r + 1)$, there is some register that is covered by at least $r + 2 = r - k + 3$ processes.

Induction Step: Let $1 \leq k \leq r - 1$ and suppose the claim is true for k. Let C be a reachable configuration in which there is a set $\mathcal{R}$ of k registers that are each covered by $r - k + 3 \geq 3$ processes. Let $\mathcal{P}_1, \ldots, \mathcal{P}_{r-k+3}$ be disjoint sets that each cover $\mathcal{R}$ with $|\mathcal{P}_i| = k$ for all i.

Divide the $n - (r - k + 3)k$ processes not in $\mathcal{P}_1 \cup \cdots \cup \mathcal{P}_{r-k+3}$ into two sets, $\mathcal{U}_1$ and $\mathcal{U}_2$, each containing at least $\lfloor (n - (r - k + 3)k)/2 \rfloor$ processes. Let $\mathcal{S}_1 = \mathcal{P}_1 \cup \mathcal{U}_1$ and $\mathcal{S}_2 = \mathcal{P}_2 \cup \mathcal{U}_2$. Then $\mathcal{S}_1 \subseteq \overline{\mathcal{P}_2 \cup \mathcal{P}_3}$ and $\mathcal{S}_2 \subseteq \overline{\mathcal{P}_1 \cup \mathcal{P}_3}$ are disjoint. Since $|\mathcal{P}_3| = k \geq 1$, there is a process $q \in \mathcal{P}_3 - (\mathcal{S}_1 \cup \mathcal{S}_2)$. For $i = 1, 2$, let C_i be the configuration obtained from C by having the processes in $\mathcal{P}_i$ do a block write. By Lemma 1, there exists $i \in \{1, 2\}$ such that every $\mathcal{S}_i$-only execution starting from C_i that contains a complete GETTS writes to a register not in $\mathcal{R}$.

Let $m = |\mathcal{S}_i|$. We inductively define a sequence of solo executions $\alpha_1, \alpha_2, \ldots, \alpha_m$ by each of the processes of $\mathcal{S}_i$ such that $\alpha_1 \cdot \alpha_2 \cdots \alpha_m$ is a legal execution from C_i that does not write to any registers outside $\mathcal{R}$ and each process covers a register not in $\mathcal{R}$. Let $1 \leq j \leq m$. Assume that $\alpha_1, \ldots, \alpha_{j-1}$ have already been defined and satisfy the claim. Consider the $\mathcal{S}_i$-only execution $\delta = \alpha_1 \cdot \alpha_2 \cdots \alpha_{j-1} \cdot \alpha$ from C_i, where α is a solo execution by another process $p_j \in \mathcal{S}_i$ that contains a complete GETTS operation. Then δ must include a write by p_j to a register outside $\mathcal{R}$ during α. Let α_j be the prefix of α up to, but not including, p_j's first write outside of $\mathcal{R}$. This has the desired properties.

Let C' be the configuration reached from C_i by performing the execution $\alpha_1 \cdot \alpha_2 \cdots \alpha_m$. Then at C', each process in $\mathcal{S}_i$ covers one of the $r-k$ registers not in $\mathcal{R}$ and $|S_i| \geq k + \lfloor (n-(r-k+3)k)/2 \rfloor \geq ((2r)^2-(r-k+1)k)/2 > (r-k)(r-k+1)$, since $2r > 2r-k, r-k+1 > 0$. Thus, by the pigeonhole principle, some register R not in $\mathcal{R}$ is covered by at least $r-k+2$ processes. Let $\mathcal{R}' = \mathcal{R} \cup \{R\}$. Each register in $\mathcal{R}$ is covered by one process from each of $\mathcal{P}_3, \ldots, \mathcal{P}_{r-k+3}$ and $\mathcal{P}_{3-i}$. Thus, each of the $k+1$ registers in $\mathcal{R}'$ is covered by $r-k+2$ processes in the configuration C', proving the claim for $k+1$.

By induction, there is a reachable configuration in which all r registers are covered by three processes each. By Lemma 1, there is an execution in which a process writes to some other register. This is impossible. □

The first paragraph of the proof also shows that, if a timestamp algorithm uses only single-writer registers, then at most one process never writes and, hence, at least $n-1$ single-writer registers are necessary.

4.2 A Tight Space Lower Bound for Static Algorithms using Nowhere Dense Universes

We now turn to the special case where timestamps come from a nowhere dense partial order, and Compare operations can be resolved using that order, without accessing shared memory. The following theorem provides a tight lower bound, since it matches a standard timestamp algorithm [17].

Theorem 3. *Any static obstruction-free timestamp algorithm that uses a nowhere dense partially ordered universe of timestamps requires at least n registers.*

Proof. We prove by induction that, for $0 \leq i \leq n$, there is a reachable configuration C_i in which a set $\mathcal{P}_i$ of i processes covers a set $\mathcal{R}_i$ of i different registers. Then, in configuration C_n, there are processes poised at n different registers.
Base Case ($i = 0$): Let C_0 be the initial configuration and let $\mathcal{P}_0 = \mathcal{R}_0 = \emptyset$.
Inductive Step: Let $1 \leq i \leq n$. Assume $C_{i-1}, \mathcal{R}_{i-1}$ and $\mathcal{P}_{i-1}$ satisfy the claim.

If $i = 1$, let p be any process. Otherwise, let $p \in \mathcal{P}_{i-1}$. Consider an execution α that starts from C_{i-1} with a block write by the processes in $\mathcal{P}_{i-1}$ to the registers of $\mathcal{R}_{i-1}$, followed by a solo execution by p in which p completes its pending operation, if any, and then performs GetTS, returning some timestamp t. Let q be a process not in $\mathcal{P}_{i-1} \cup \{p\}$. We show that a solo execution by q, starting from C_{i-1}, in which it performs an infinite sequence of GetTS operations must eventually write to a register not in $\mathcal{R}_{i-1}$. Let t_j be the timestamp returned by the j'th instance of GetTS by q in this solo execution. Then $t_j < t_{j+1}$ for all $j \geq 1$. Since $\{j \in \mathbf{N} \mid t_1 < t_j < t\}$ is finite, there exists $j \in \mathbf{N}$ such that $t_j \not< t$.

Suppose that q does not write to any register outside $\mathcal{R}_{i-1}$ during the solo execution, β, of j instances of GetTS, starting from C_{i-1}. Then $\beta \cdot \alpha$ is indistinguishable from α to p, so p returns t as the result of its last GetTS in

Code for process p_i (for $1 \leq i \leq n-1$):
```
GetTS
    $t \leftarrow \max(\textsc{Collect}(R)) + 1$
    $R[i] \leftarrow t$
    return $(t, 0)$
```

Code for process p_n:
```
GetTS
    $t \leftarrow \max(\textsc{Collect}(R))$
    if $t > \mathit{oldt}$ then $c \leftarrow 0$
    $c \leftarrow c + 1$
    $\mathit{oldt} \leftarrow t$
    return $(t, c)$
```

Fig. 1. An eponymous algorithm using $n-1$ registers.

$\beta \cdot \alpha$. Therefore, $t_j < t$. This contradicts the definition of j, so q must write outside $\mathcal{R}_{i-1}$. Consider the solo execution of q starting from C_{i-1} until it first covers some register R outside $\mathcal{R}_{i-1}$. Let C_i be the resulting configuration. Then $\mathcal{P}_i = \mathcal{P}_{i-1} \cup \{q\}$ and $\mathcal{R}_i = \mathcal{R}_{i-1} \cup \{R\}$ satisfy the claim for i. $\square$

Jayanti, Tan and Toueg proved that linearizable implementations of perturbable objects require at least $n-1$ registers [16]. Roughly speaking, an object is perturbable if some sequence of operations on the object by one process must be visible to another process that starts executing later. General timestamps do not have this property. However, the proof technique of [16] can be applied to the special case considered in Theorem 3 (even though timestamps are not linearizable). The proof technique used in Theorem 3 is similar to theirs, but is considerably simpler, and gives a slightly stronger lower bound. Although our improvement to the bound is small, it is important, since it proves a complexity separation, showing that using nowhere dense sets of timestamps requires more registers than used by the algorithm of the next section.

5 An Eponymous Algorithm

In this section, we show that there is a simple wait-free eponymous algorithm that uses only $n-1$ single-writer registers, which is optimal. The timestamps generated will be ordered pairs of non-negative integers, ordered lexicographically. This shows that the lower bound in Sect. 4.2 is not true for all domains.

The algorithm uses an array $R[1..n-1]$ of single-writer registers, each initially 0. Processes $p_1, \ldots, p_{n-1}$ use this array to collaboratively create the first component of the timestamps by the simple method [17] discussed in Sect. 1. The second component of any timestamp they generate is 0. The last process, p_n, reads the registers of the other processes to determine the first component of its timestamp, and produces the values for the second component of its timestamp on its own. Process p_n does not write into shared memory.

The implementation of GetTS is presented in Figure 1. In the code for p_n, *oldt* and c are persistent variables, initially 0. $\textsc{Compare}((t_1, c_1), (t_2, c_2))$ returns true if and only if either $t_1 = t_2$ and $c_1 < c_2$ or $t_1 < t_2$. The value stored in each component of R does not decrease over time. So, if two non-concurrent Collects are performed on R, the maximum value seen by the later Collect will be at least as big as the maximum value seen by the earlier Collect.

Theorem 4. *Figure 1 gives a timestamp algorithm using $n-1$ registers with step complexity $O(n)$.*

Proof. Suppose an instance, I_1, of GetTS returns (t_1, c_1) before the invocation of another instance I_2 of GetTS, returns (t_2, c_2). We show that Compare$((t_1, c_1), (t_2, c_2))$ returns true. We consider several cases.
Case 1: I_1 and I_2 are both performed by p_n. It follows from the code that p_n generates an increasing sequence of timestamps (in lexicographic order): each time p_n produces a new timestamp, it either increases the first component or leaves the first component unchanged and increases the second component.
Case 2: p_n performs I_2 but some process $p_i \neq p_n$ performs I_1. During I_2, the value p_n sees when it reads $R[i]$ is at least t_1, so $t_2 \geq t_1$. Furthermore, $c_2 \geq 1 > 0 = c_1$.
Case 3: I_2 is not performed by p_n. Then t_1 was the value of some component of R some time before the end of I_1 (because it was either read by p_n while performing I_1, or was written by another process while performing I_1). The value of this component of R is at least t_1 when I_2 reads it, so $t_2 \geq t_1 + 1$.

In all three cases, a Compare$((t_1, c_1), (t_2, c_2))$ will return true, as required. Since R has $n-1$ components, the step complexity of GetTS is $O(n)$. $\square$

6 Anonymous Algorithms

We present two new anonymous timestamp algorithms. The first uses n registers and, as we shall see in Sect. 7, it is space-optimal. However, this algorithm, like Guerraoui and Ruppert's algorithm [12], is not bounded wait-free. The second algorithm uses $O(n^2)$ registers, but it is bounded wait-free. It is an open question whether there is a bounded wait-free algorithm that uses $O(n)$ registers.

6.1 A Wait-Free Algorithm Using n Registers

The first algorithm uses an array $A[1..n]$ of registers, each initially 0. The timestamps are non-negative integers. Before a process returns a timestamp, it records it in A so that subsequent GetTS operations will see the value and return a larger one. We ensure this by having a process choose its timestamp by reading all timestamps in A and choosing a larger one. The anonymity of the algorithm presents a challenge, however. In a system with only registers, two processes running in lockstep, performing the same sequence of steps, have the same effect as a single process: there is no way to tell these two executions apart. Even the two processes themselves cannot detect the presence of the other. Consider an execution where some process p takes no steps. We can construct another execution where p runs as a *clone* of any other process q, and p stops taking steps at any time, covering any register that q wrote to. Thus, at any time, a clone can overwrite any value written in a register (except the first such value) with an older value. In the timestamp algorithm, if the value t chosen by one process and recorded in A is overwritten by values smaller than t, another process that begins performing GetTS after the value t has been chosen could again choose t as a timestamp, which would be incorrect.

```
GetTS
    $t \leftarrow \max(\text{Collect}(A)) + 1$
    for $i \leftarrow 1..M(t)$
        for $j \leftarrow 1..n$
            if $A[j] < t$ then $A[j] \leftarrow t$
        end for
    end for
    return($t$)
```

Fig. 2. A wait-free anonymous timestamp algorithm using n registers.

To avoid this problem, we ensure that the evidence of a timestamp cannot be entirely overwritten after GetTS returns it. We say that a value v is *established* in configuration C if there exists a shared register that, in every configuration reachable from C, contains a value larger than or equal to v. (Note that, if a value larger than v is established, then v is also established.) Once a value v is established, any subsequent GetTS can perform a Collect of the registers and see that it should return a value greater than v. Thus, our goal is to ensure that values are established before they are returned by GetTS operations.

The algorithm, shown in Fig. 2, uses several measures to do this. The first is having processes read a location before writing it and never knowingly overwrite a value with a smaller value. This implies a value in a register is established whenever there are no processes covering it, poised to write smaller values. This measure alone is insufficient: if p writes to a register between q's read and write of that register, q may overwrite a larger value with a smaller one. However, it limits the damage that a process can do. Another measure is for GetTS to record its output in many locations before terminating. It also writes to each of those locations repeatedly, using a larger number of repetitions as the value of the timestamp gets larger. The number of repetitions, $M(t)$, that GetTS uses to record the timestamp t, is defined recursively by $M(1) = 1$ and $M(t) = n(n-1)\sum_{i=1}^{t-1} M(i)$ for $t > 1$. Solving this recurrence yields $M(t) = n(n-1)(n^2-n+1)^{t-2}$ for $t > 1$. The Compare(t_1, t_2) algorithm simply checks whether $t_1 < t_2$. Correctness follows easily from the following lemma.

Lemma 5. *Whenever* GetTS *returns a value t, the value t is established.*

Theorem 6. *Figure 2 gives a wait-free anonymous timestamp algorithm using n registers.*

When GetTS returns t, it performs $\Theta(n^{2t-1})$ steps. Thus, the algorithm is wait-free, but not bounded wait-free. In an execution with k GetTS operations, all timestamps are at most k, since GetTS can choose timestamp t only if another (possibly incomplete) GetTS has chosen $t-1$ and written it into A. Each of the n registers must contain enough bits to represent one timestamp.

6.2 A Bounded Wait-Free Algorithm Using $O(n^2)$ Registers

The preceding algorithm is impractical because of its time complexity. Here, we give an algorithm that runs in polynomial time and space. As in the pre-

```
GetTS
    $t \leftarrow \max(\max(\textsc{Collect}(A), t) + 1$
    $row \leftarrow t \bmod (2n-1)$
    for $i \leftarrow 1..n$
        $A[row, i] \leftarrow t$
        if $\max(\textsc{Collect}(A)) \geq t + n - 1$ then return($t$)
    end for
    return($t$)
```

Fig. 3. A bounded wait-free anonymous timestamp algorithm using $O(n^2)$ registers.

ceding algorithm, timestamps are non-negative integers and a process chooses a timestamp that is larger than any value recorded in the array A. However, now, $A[0..2n-2, 1..n]$ is a two-dimensional array of registers and the method for recording a chosen value in A is quite different. Before a process p returns a timestamp t, it writes t into the entries of one row of the array, chosen as a function of t. A careful balance must be maintained: p should not write too many copies of t, because doing so could overwrite information written by other, more advanced processes, but p must write enough copies to ensure that t is not expunged by other, less advanced processes.

Process p attempts to write t into all entries of one row, but stops writing if it sees value $t+n-1$ or larger anywhere in the array. We show that, if this occurs, then another process q has already returned a timestamp larger than t. (In that case, q will have already ensured that no future GetTS will ever return a value smaller than its own timestamp, so p can safely terminate and return t.) This avoids the problem of p writing too many copies of t.

To avoid the problem of p writing too few copies of t, the rows are chosen in a way that ensures that one value cannot be overwritten by another value unless those two values are sufficiently far apart. This ensures that other processes will terminate before obliterating all evidence of the largest timestamp written in A.

The algorithm is presented in Fig. 3. In addition to the shared array A, each process has a persistent local variable t, initialized to 0. Again, Compare(t_1, t_2) is performed by simply checking whether $t_1 < t_2$.

We remark that, if a value $v > 0$ is written into A, then $v-1$ appeared in A earlier. The correctness of the algorithm follows easily from the lemma below.

Lemma 7. *Whenever* GetTS *returns a value t, the value t is established.*

Proof. We prove the lemma by induction on the number of return events.
Base case: If no return events have occurred, the lemma is vacuously satisfied.
Induction step: Let $k > 0$. Assume that, at each of the first $k-1$ return events, the returned value is established.

Consider the configuration C just after the kth return event, in which process p returns t. We show t is established in C by considering two cases, depending on the termination condition that p satisfies.
Case 1: Suppose p returns t because it saw some value $m \geq t+n-1$ in A.

Some process wrote m before p read it. It follows that each of the values $t, t+1, t+2, \ldots, t+n-1, \ldots, m$ appeared in A at some time during the execution before C. For $1 \leq i \leq n-1$, let p_i be the process that first wrote the value $t+i$ into A. These processes do not include p, since p returns t at configuration C. If all of these $n-1$ processes are distinct, then no process will ever write a value smaller than t after C, so t is established. Otherwise, by the pigeonhole principle, $p_i = p_j$ for some $i < j$. Process p_i must have completed the instance of GetTS that first wrote $t+i$ before it began the instance of GetTS that first wrote $t+j$. The former instance returns $t+i$, so the value $t+i$ is established when it is returned, by the induction hypothesis. Thus, in C, the value $t+i$ is established and, hence, so is the value t.

Case 2: Suppose p terminates after it has completed all n iterations of the loop.

If $t < 2n-1$, in the first loop iteration of the GetTS that returns t, p writes t into $A[t, 1]$. No value smaller than t can ever be written there, so t is established.

Now assume $t \geq 2n-1$. The values $t-1, t-2, \ldots, t-n$ were written into A prior to the completion of p's first Collect. For $0 \leq i < n$, let p_i be the process that first wrote the value $t-n+i$ into A. If $p_i = p$ for some i, then p returned $t-n+i$ before starting the instance of GetTS that returned t, and the value $t-n+i$ is established, by the induction hypothesis. Otherwise, by the pigeonhole principle, we must have $p_i = p_j$ for some $0 \leq i < j < n$. When process p_i first wrote $t-n+i$, it returns $t-n+i$, that value is established, by the induction hypothesis. In either case, some value greater than or equal to $t-n$ is established by the time that p completes its first Collect. Hence, $t-n$ is also established.

We show no process writes values smaller than t in row $t \bmod (2n-1)$ more than once after p's first write of t. Suppose not. Let q be the process that first does a second such write. Suppose the first such write by q writes the value $t_1 < t$ and the second writes the value $t_2 < t$. Then $t_1 \leq t-(2n-1)$, since $t_1 \bmod (2n-1) = t \bmod (2n-1)$ and $t_1 < t$. Similarly, $t_2 \leq t-(2n-1)$. When q performs Collect just after it writes t_1, it sees a value $t-n$ or larger in A, since $t-n$ is established. Furthermore, $t-n \geq (t_1+2n-1)-n = t_1+n-1$ and the loop terminates. So, when q writes t_2, that write is part of a different instance of GetTS. Again, when q performs Collect in the first line of that instance of GetTS, it must see a value $t-n$ or larger, since $t-n$ is established. Thus, $t_2 \geq t-n+1$, contradicting the fact that $t_2 \leq t-2n$.

Thus, when p returns t, it has written the value t into all n entries of row $t \bmod (2n-1)$ of A and at most $n-1$ of those copies are subsequently overwritten by smaller values. So, t is established. □

The worst-case running time of GetTS is $O(n^3)$, since each Collect takes $\Theta(n^2)$ steps. Timestamps are bounded by the number of GetTS operations invoked, and each register must be large enough to contain one timestamp.

Theorem 8. *Figure 3 gives a wait-free anonymous timestamp algorithm using $O(n^2)$ registers with step complexity $O(n^3)$.*

7 A Tight Space Lower Bound for Anonymous Algorithms

The anonymous timestamp algorithm given in Sect. 6.1 uses n registers. In it, a process may write its timestamp value to each of the n registers. Intuitively, this is done to ensure that other processes, which could potentially cover $n-1$ of the registers, cannot overwrite all evidence of the timestamp. Here, we sharpen this intuition into a proof that at least n registers are required for anonymous timestamp algorithms. This applies to obstruction-free implementations of timestamps (and therefore to wait-free implementations).

Lemma 9. *Let $n \geq 2$. In any anonymous obstruction-free timestamp implementation for n processes, a solo execution of $k \leq n$ instances of* GetTS*, starting from an initial configuration, writes to at least k different registers.*

Proof. Suppose not. Consider the smallest k such that there is a solo execution of $k \leq n$ instances of GetTS by a process p, starting from an initial configuration, which writes to a set $\mathcal{R}$ of fewer than k different registers. Let α be the prefix of this execution consisting of the first $k-1$ instances of GetTS. By definition of k, it writes to at least $k-1$ different registers. Thus, $|\mathcal{R}| = k-1$ and $\mathcal{R}$ is the set of registers written to during α. Let C be the configuration immediately after the last write in α (or the initial configuration, if there are no writes in α).

We define another execution β. First, add clones of p to execution α, such that one clone continues until just before p last writes to each register in $\mathcal{R}$. Let q be the last of these clones to take a step. (If $\mathcal{R}$ is empty, then let q be any process other than p.) Then p performs one more instance of GetTS after those it performed in α. Let t be the value returned by this operation. Note that p only writes to registers in $\mathcal{R}$. Next, let the clones do a block write to $\mathcal{R}$. Let C' be the configuration immediately after the block write. Finally, q runs solo to complete its operation, if necessary, and then does one more GetTS.

Each register has the same value in configurations C and C' and p's state in C is the same as q's state in C'. Thus, q's steps after C' will be identical to p's steps after C, and q's last GetTS will also return t. This is a contradiction, since that operation begins after p's last GetTS, which also returned t, ended.

Theorem 10. *Any n-process anonymous obstruction-free timestamp algorithm uses at least n registers.*

Acknowledgements We thank Rachid Guerraoui for helpful discussions. Funding was provided by the Natural Sciences and Engineering Research Council of Canada and by the Scalable Synchronization Group at Sun Microsystems.

References

[1] Karl Abrahamson. On achieving consensus using a shared memory. In *Proc. 7th ACM Symposium on Principles of Distributed Computing*, pages 291–302, 1988.

[2] Yehuda Afek, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. A bounded first-in, first-enabled solution to the l-exclusion problem. *ACM Transactions on Programming Languages and Systems*, 16(3):939–953, May 1994.

[3] Hagit Attiya and Arie Fouren. Algorithms adapting to point contention. *Journal of the ACM*, 50(4):444–468, July 2003.

[4] James Burns and Nancy Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993.

[5] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1):11–16, July 1991.

[6] Danny Dolev and Nir Shavit. Bounded concurrent time-stamping. *SIAM Journal on Computing*, 26(2):418–455, April 1997.

[7] Cynthia Dwork, Maurice Herlihy, Serge Plotkin, and Orli Waarts. Time-lapse snapshots. *SIAM Journal on Computing*, 28(5):1848–1874, 1999.

[8] Cynthia Dwork and Orli Waarts. Simple and efficient bounded concurrent timestamping and the traceable use abstraction. *Journal of the ACM*, 46(5):633–666, September 1999.

[9] Panagiota Fatourou, Faith Ellen Fich, and Eric Ruppert. Time-space tradeoffs for implementations of snapshots. In *Proc. 38th ACM Symposium on Theory of Computing*, pages 169–178, 2006.

[10] Faith Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *Journal of the ACM*, 45(5):843–862, September 1998.

[11] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, August 1991.

[12] Rachid Guerraoui and Eric Ruppert. Anonymous and fault-tolerant shared-memory computing. *Distributed Computing.* To appear. A preliminary version appeared in *Distributed Computing, 19th International Conference*, pages 244–259, 2006.

[13] Sibsankar Haldar and Paul Vitányi. Bounded concurrent timestamp systems using vector clocks. *Journal of the ACM*, 49(1):101–126, January 2002.

[14] Amos Israeli and Ming Li. Bounded time-stamps. *Distributed Computing*, 6(4):205–209, 1993.

[15] Amos Israeli and Meir Pinhasov. A concurrent time-stamp scheme which is linear in time and space. In *Proc. 6th Int. Workshop on Distributed Algorithms*, pages 95–109, 1992.

[16] Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for nonblocking implementations. *SIAM Journal on Computing*, 30(2):438–456, 2000.

[17] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[18] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[19] Ming Li, John Tromp, and Paul M. B. Vitányi. How to share concurrent wait-free variables. *Journal of the ACM*, 43(4):723–746, July 1996.

[20] Friedemann Mattern. Virtual time and global states of distributed systems. In *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.

[21] Marios Mavronicolas, Loizos Michael, and Paul Spirakis. Computing on a partially eponymous ring. In *Proc. 10th International Conference on Principles of Distributed Systems*, pages 380–394, 2006.

[22] Paul M. B. Vitányi and Baruch Awerbuch. Atomic shared register access by asynchronous hardware. In *Proc. 27th IEEE Symposium on Foundations of Computer Science*, pages 233–243, 1986.