On the time and space complexity of randomized test-and-set

George Giakkoupis, Philipp Woelfel

On the Time and Space Complexity of Randomized Test-And-Set

(Extended Abstract)

George Giakkoupis[*] [†]
INRIA Rennes-Bretagne Atlantique
Campus Universitaire de Beaulieu
35042 Rennes Cedex, France
george.giakkoupis@inria.fr

Philipp Woelfel[†]
Dept. of Computer Science
University of Calgary
Calgary, AB T2N 1N4, Canada
woelfel@ucalgary.ca

ABSTRACT

We study the time and space complexity of randomized Test-And-Set (TAS) implementations from atomic read/write registers in asynchronous shared memory models with n processes. We present an adaptive TAS algorithm with an expected (individual) step complexity of $O(\log^* k)$, for contention k, against the oblivious adversary, improving a previous (non-adaptive) upper bound of $O(\log \log n)$ (Alistarh and Aspnes, 2011). We also present a modified version of the adaptive RatRace TAS algorithm (Alistarh et al., 2010), which improves the space complexity from $O(n^3)$ to $O(n)$, while maintaining logarithmic expected step complexity against the adaptive adversary. We complement this upper bound with an $\Omega(\log n)$ lower bound on the space complexity of any TAS algorithm that has the nondeterministic solo-termination property (which is a weaker progress condition than wait-freedom). No non-trivial lower bounds on the space requirements of TAS were known prior to this work.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

Keywords

Test-And-Set, Leader Election, shared memory, randomization, strong/weak adversary, time/space complexity

[*]Supported by the Pacific Institute for the Mathematical Sciences (PIMS)

[†]Supported by a Discovery Grant from the Natural Sciences and Research Council of Canada (NSERC)

PODC'12, July 16–18, 2012, Madeira, Portugal.

1. INTRODUCTION

In this paper we study the (randomized) time and space complexity of *Test-And-Set (TAS)* implementations from atomic registers in asynchronous shared memory systems with n processes. The TAS object is a fundamental synchronization primitive. It has been used in algorithms for classical problems such as mutual exclusion and renaming [3, 9].

A TAS object stores a bit, whose value is initially 0. It allows a single operation, `TAS()`, which sets the bit and returns its previous value. TAS objects are among the simplest natural primitives which have no deterministic wait-free linearizable implementations from atomic registers, even in systems with only two processes. In fact, in systems with two processes, a consensus protocol can be implemented deterministically from a TAS object and vice versa.

For randomized shared-memory algorithms, *adversary models* are used to describe how the scheduling is influenced by the random decisions made by processes. The strongest reasonable adversary is the *adaptive adversary*, which bases scheduling decisions on the entire past history of events including coin flips made by processes. An efficient randomized implementation of a TAS object from $O(n)$ registers dates back to 1992: Afek, Gafni, Tromp, and Vitányi [1] gave an algorithm with an expected (individual) step complexity of $O(\log n)$, which means that the maximum number of steps taken by any process has expectation $O(\log n)$. Recently, Alistarh, Attiya, Gilbert, Giurgiu, and Guerraoui [3] presented an *adaptive* algorithm called *RatRace*, in which the expected step complexity is logarithmic in k, the total number of processes accessing the TAS object. The space requirements of RatRace are higher, though, as $\Theta(n^3)$ registers are needed.

Even though non-trivial lower bounds are not known, no TAS algorithm with a sub-logarithmic expected step complexity (against the adaptive adversary) has been found, yet. The adaptive adversary, however, seems to be too strong in many cases to model realistic system behavior. Motivated by the fact that consensus algorithms benefit from weaker adversary models, Alistarh and Aspnes [2] devised a very simple but elegant TAS algorithm with an expected step complexity of $O(\log \log n)$ for the *oblivious adversary model*, where the adversary has to make all scheduling decisions at the beginning of the execution. In the following we denote this algorithm by *AA-algorithm*. Although not explicitly mentioned in the paper, the AA-algorithm even works for a slightly stronger adversary, the so-called *R/W-oblivious ad-*

versary. An R/W-oblivious adversary can take all past operations, including coin-flips, of processes into account when making scheduling decisions, but it cannot see whether a process will read or write in its next step, if that decision is made by the process at random. Since the AA-algorithm uses RatRace, its space complexity is also dominated by the $\Theta(n^3)$ registers from the RatRace implementation.

Motivated by their result, Alistarh and Aspnes asked whether any better TAS algorithm exists for the oblivious or even stronger adversary models. We answer this question in the affirmative: We present an adaptive algorithm that has expected step complexity of $O(\log^* k)$ against the oblivious adversary, where k is the maximum contention. In fact, our result holds for the slightly stronger *location-oblivious adversary.* This adversary makes scheduling decisions based on all past events (including coin-flips), but it does not know which registers processes will access in their next step, if those decisions are made at random. Further, our algorithm is the first one with sub-logarithmic expected step complexity that needs only $O(n)$ registers.

However, our algorithm is not efficient against the R/W-oblivious adversary. Instead, we present a modification of the AA-algorithm that needs only $O(n)$ registers and is adaptive, i.e., its expected step complexity is $O(\log \log k)$ if the contention is bounded by k.

The AA-algorithm has the nice property that its performance degrades gracefully when the adversary is not R/W-oblivious, and against the adaptive adversary it still achieves an expected step complexity of $O(\log n)$. This is a desirable property, as one does not need to rely on the system not to behave like an adaptive adversary. In its simple form, our adaptive algorithm with expected step complexity $O(\log^* k)$ does not exhibit this behavior—an adaptive adversary can find a schedule where processes need $\Omega(k)$ steps to complete their TAS operation (where k is the maximum contention). However, we present a general method to combine any TAS algorithm A with RatRace, so that the combined algorithm inherits the "best" complexity from both models. I.e., if A has expected step complexity $C(k)$ against an R/W-oblivious or location-oblivious adversary, then the combined algorithm has expected step complexity $O(C(k))$ against the same adversary, while at the same time it achieves $O(\log k)$ expected step complexity if the adversary is adaptive.

We complement our algorithms with lower bounds. First, we show that at least $\Omega(\log n)$ registers are needed for any randomized TAS implementation (from atomic registers) that satisfies the progress condition *nondeterministic solo-termination* [10], which is strictly weaker than wait-freedom. This is the first non-trivial lower bound on the space complexity of randomized TAS implementations.

Finally, we show for any randomized TAS implementation for two processes, that the oblivious adversary can schedule processes in such a way that for any $t > 0$ with probability at least $1/4^t$ one of the processes does not finish its `TAS()` method call within fewer than t steps. This result immediately implies the same lower bound on 2-process consensus, complementing a lower bound by Attiya and Censor-Hillel [6]. The authors showed for some constant c that with probability at least $1/c^t$ any randomized f-resilient n-process consensus algorithm does not terminate within $t(n-f)$ steps. However, their lower bound proof only works for $n \geq 3$ processes. Our lower bound thus fills in the missing 2-process case.

Preliminaries

We consider an asynchronous shared memory model where up to n processes communicate by reading and writing to shared atomic multi-reader multi-writer registers. Processes may fail by crashing at any time. Algorithms are randomized and can use local coin-flips to make random decisions. The scheduling and process crashes are controlled by an adversary, who at any point of an execution decides which process will take the next step. An adversary is *adaptive* if a scheduling decision is based on the entire past execution, including the results of local coin-flips. An adversary is *location-oblivious* [4] if a scheduling decision is based on any shared memory steps processes have taken in the past, and the type and argument of pending write operations (but not the register on which the operation will occur). An adversary is *R/W-oblivious* if a scheduling decision is based on any shared memory steps processes have taken in the past, and the locations of pending shared memory operations. While the R/W-oblivious adversary "knows" which register R a process is going to access in its next step, it cannot take into account whether the process reads or writes R.

We say that a randomized algorithm A has *expected step complexity* t, if for the random execution obtained by a scheduling of the "worst" adversary from a given set of adversaries, the maximum number of steps taken by any process has expectation t. The contention of an execution, usually denoted k, is the maximum number of processes taking at least one step in that execution. If the expected step complexity t is a function of k (instead of n) then we say that the algorithm is adaptive.

A Leader Election object, `LeaderElect`, provides a method `elect()`, which every process can call at most once and which returns a binary value. If some processes call `elect()` then at most one such call can return the value `True`, and if no process crashes then exactly one `elect()` call returns `True`. If a process' `elect()` call returns `True` or `False`, then we say the process *wins* resp. *loses* the Leader Election. It is not hard to see that any (randomized or deterministic) implementation of a `LeaderElect` object can be used together with one shared atomic register to implement a linearizable TAS object [11]. In this implementation, a `TAS()` method call consists of at most one call of `elect()` and in addition one read- and possibly one write-operation. Therefore, in this paper we will implement `LeaderElect` objects, and we don't have to worry about linearizability.

In order to implement Leader Election, we use several other objects as building blocks. One building block is a randomized 2-process `LeaderElect` object which uses only a constant number of registers, and where the `elect()` method has constant expected step complexity. Such an implementation was proposed by Tromp and Vitányi [13]. Another building block is a (deterministic) splitter object [12], `Splitter`, which provides the method `split()`. The method takes no parameters and returns a value in $\{L, R, S\}$. If a process' `split()` call returns S, we say that the process *wins* the splitter. If k processes call `split()`, at most $k-1$ receive the return value L, at most $k-1$ receive the value R, and at most one receives the value S. Thus, if only one process calls `split()`, the method returns S. A *randomized* splitter object [7], `RSplitter`, has the last two properties above, i.e., at most one `split()` call returns S, and if `split()` is called only once then it returns S. But now, if a `split()` call does not return S, it returns L or R independently with proba-

bility 1/2—thus it is possible that all calls return the same value in $\{L, R\}$. Deterministic and randomized splitters can both be implemented from $O(1)$ atomic registers such that any call to `split()` takes only a constant number of steps.

2. FAST LEADER ELECTION

In Section 2.1, we introduce the Group Election primitive, and present an implementation of a Leader Election object from Group Election objects. Then, in Sections 2.2 and 2.3, we describe randomized implementations of Group Election from registers, for the location-oblivious and the R/W-oblivious adversary models.

2.1 Leader Election from Group Election

A Group Election object, `GroupElect`, provides the method `elect()`, which takes no parameters and returns either `True` or `False`. We say that the processes whose `elect()` call return `True` get *elected*. If some processes call `elect()`, then at least one process must get elected. The performance of a `GroupElect` object is measured by the total number of processes that get elected. We define the *performance parameter* of a `GroupElect` object to be the smallest function $f : \{1, \ldots, n\} \to [1, n]$ such that the *expected* number of processes that get elected is at most $f(k)$, when $k \in \{1, \ldots, n\}$ processes call `elect()`.

We now describe an implementation of a `LeaderElect` object from n `GroupElect` objects $GE_1, \ldots, GE_n$. The implementation uses also n `Splitter` objects $SP_1, \ldots, SP_n$, and n 2-process `LeaderElect` objects $LE_1, \ldots, LE_n$. Each process participates in a series of Group Elections on $GE_1, GE_2, \ldots$ If a process does not get elected in one of the Group Elections it participates in, then it immediately loses the implemented Leader Election. If a process gets elected in the Group Election on GE_i, then it tries to win splitter SP_i. If its SP_i.`split()` call returns L, then the process loses the Leader Election. If it returns R, then the process continues with the next Group Election, on GE_{i+1}. Finally, if the process wins SP_i, then it does not participate in other Group Elections, instead it participates in a series of 2-process Leader Elections, on objects $LE_i, LE_{i-1}, \ldots, LE_1$, until it either loses one of these elections or wins all of them. In the latter case, it wins the implemented `LeaderElect` object, and otherwise it loses it.

The proof of correctness of the implementation is straightforward, so we just present a sketch. First observe that at most $n-i+1$ processes call GE_i.`elect()`, and thus no more than n `LeaderElect` objects are needed: If $j > 0$ processes call GE_i.`elect()`, then at most j processes call SP_i.`split()`, at most $j-1$ of them receive the value R, and thus at most $j-1$ call GE_{i+1}.`elect()`. Next we observe that each 2-process `LeaderElect` object LE_i is indeed accessed by at most two processes: the winner of SP_i and the winner of LE_{i+1} (if $i < n$). The winner of the implemented `LeaderElect` object is the winner of LE_1, thus at most one process wins `LeaderElect`. Finally, if no process crashes then some process wins: To show this we use that at least one of the processes that call GE_i.`elect()` gets elected, and at least one of the processes that call SP_i.`split()` receives a return value other than L.

To bound the step complexity of the implementation we will use the following terminology. Let $M = (M_0, M_1, \ldots)$ be a Markov chain with state space $\{0, \ldots, n\}$ that is non-increasing. The *rate* of M is the function $r : \{1, \ldots, n\} \to [0, n]$ such that $r(j) = \mathbf{E}[M_{i+1} \mid M_i = j]$, for $1 \le j \le n$. For any non-decreasing $r : \{1, \ldots, n\} \to [0, n]$, we denote by $\Delta_r(n)$ the maximum (expected) hitting time $h_{n,0}$ over all non-increasing Markov chains on $\{0, \ldots, n\}$ with rate at most r.

Class `GroupElect`

```
/* Let ℓ = ⌈log n⌉                                   */
shared:
    int R[1 ... (ℓ + 1)] = [0 ... 0]
    int flag = 0
```

Method `elect()`

```
1 if flag.Read() = 1 return False
2 flag.Write(1)
3 Choose an integer x ∈ {1, ..., ℓ} independently at
  random s.t. Pr(x = i) = 1/2^i, for 1 ≤ i < ℓ, and
  Pr(x = ℓ) = 1/2^(ℓ−1)
4 R[x].Write(1)
5 if R[x + 1].Read() = 0 return True
6 return False
```

Figure 1: Group Election implementation for the location-oblivious adversary.

Suppose now that the performance parameter of `GroupElect` objects GE_i is bounded by a non-decreasing function f. Let N_i be the number of processes that call GE_i.`elect()`, and let $i^* = \max\{i \colon N_i > 0\}$ be the total number of Group Elections executed. (Clearly, i^* is also a bound on the number of 2-process Leader Elections). If $N_i > 0$ then N_{i+1} is the number of processes that get elected on GE_i minus the number of processes whose call SP_i.`split()` returns a value in $\{L, S\}$. The latter number is at least one, so $\mathbf{E}[N_{i+1} \mid N_i = j] \le f(j) - 1$, for $j > 0$. It follows that $\mathbf{E}[i^*] \le \Delta_{f-1}(k)$, and thus, the expected step complexity of the implementation is $O(\Delta_{f-1}(k))$.

Thus, the following statement holds.

LEMMA 2.1. *There is a randomized implementation of a* `LeaderElect` *object with expected step complexity* $O(\Delta_{f-1}(k))$*, from* n `GroupElect` *objects with performance parameter at most* f*, and* $O(n)$ *registers.*

2.2 Location-Oblivious Adversary

We now present a randomized implementation of a Group Election object for the location-oblivious adversary.

LEMMA 2.2. *The construction in Figure 1 implements a* `GroupElect` *object with step complexity* $O(1)$*, space complexity* $O(\log n)$*, and performance parameter* $f(k) \le 2 \log k + 6$ *against the location-oblivious adversary.*

PROOF. Let A be a location-oblivious adversary and consider a random execution E obtained by A. Let i^* be the largest index such that some process p writes to $R[i^*]$ in line 4. Then p reads values 0 from $R[i^* + 1]$ in the next line, and thus returns `True`. Hence, at least one process gets elected.

In the rest of the proof we show the upper bound on f. Let $k' \le k$ be the number of processes that read register $flag$ in line 1 before some process writes $flag$ in the next line. The number of processes elected depends only on k'

(and not on k), and k' is fixed as soon as the first process writes $flag$ (that is, before any random choices are made). Thus, we can assume w.l.o.g. that k is fixed in advanced and $k' = k$.

Let $p_1, \ldots, p_k$ be the k processes participating in E in the order in which they perform their write operation in line 4, and let $\ell_1, \ldots, \ell_k$ be the respective locations in array R that they write to, i.e., process p_j writes to register $R[\ell_j]$. Since A is location-oblivious, it does not know ℓ_j before p_j writes to $R[\ell_j]$. Thus, by the principle of deferred decisions, we can assume that only after the adversary has decided the order $p_1, \ldots, p_j$ does p_j choose ℓ_j. Let X_j be the 0/1 random variable with $X_j = 1$ if and only if p_j reads a '0' in line 5, and thus gets elected. Let Y_j be the 0/1 random variable with $Y_j = 1$ if and only if none of the processes $p_1, \ldots, p_{j-1}$ writes to register $R[\ell_j + 1]$ (i.e., $\ell_{j'} \neq \ell_j + 1$, for all $j' < j$). Clearly, $X_j \leq Y_j$. Then,

$$f(k) = \mathbf{E}\left[\sum_{1 \leq j \leq k} X_j\right] = \sum_{1 \leq j \leq k} \mathbf{E}[X_j] \leq \sum_{1 \leq j \leq k} \mathbf{E}[Y_j].$$

We have

$$\begin{aligned}
\mathbf{E}[Y_j] &= \Pr\left(\bigwedge_{1 \leq j' < j} \ell_{j'} \neq \ell_j + 1\right) \\
&= \sum_{1 \leq i \leq \ell} \Pr\left(\ell_j = i \wedge \ell_1 \neq i+1 \wedge \cdots \wedge \ell_{j-1} \neq i+1\right) \\
&= \sum_{1 \leq i \leq \ell} \Pr(\ell_j = i) \prod_{j'=1}^{j-1} \Pr(\ell_j \neq i+1) \\
&= \sum_{1 \leq i < \ell} \frac{1}{2^i}\left(1 - \frac{1}{2^{i+1}}\right)^{j-1} + \frac{1}{2^{\ell-1}}.
\end{aligned}$$

Thus,

$$\begin{aligned}
f(k) &\leq \sum_{1 \leq j \leq k}\left(\sum_{1 \leq i < \ell} \frac{1}{2^i}\left(1 - \frac{1}{2^{i+1}}\right)^{j-1} + \frac{1}{2^{\ell-1}}\right) \\
&= \sum_{1 \leq i < \ell} \frac{1}{2^i} \sum_{1 \leq j \leq k}\left(1 - \frac{1}{2^{i+1}}\right)^{j-1} + \sum_{1 \leq j \leq k} \frac{1}{2^{\ell-1}} \\
&= \sum_{1 \leq i < \ell} \frac{1}{2^i} \cdot \frac{1 - \left(1 - \frac{1}{2^{i+1}}\right)^k}{1/2^{i+1}} + \frac{k}{2^{\ell-1}} \\
&= 2\sum_{1 \leq i < \ell}\left(1 - \left(1 - \frac{1}{2^{i+1}}\right)^k\right) + \frac{k}{2^{\ell-1}}.
\end{aligned}$$

We upper-bound the sum in the last line by observing that each term is at most 1, and also the i-th term is at most $1 - \left(1 - \frac{k}{2^{i+1}}\right) = \frac{k}{2^{i+1}}$, by the known inequality $(1-\epsilon)^k \geq 1 - k\epsilon$. Thus,

$$\begin{aligned}
f(k) &\leq 2\sum_{1 \leq i < \log k} 1 + 2\sum_{\log k \leq i < \ell} \frac{k}{2^{i+1}} + \frac{k}{2^{\ell-1}} \\
&\leq 2(\log k + 1) + 2\frac{k}{2^{\log k}} + \frac{k}{2^{\ell-1}} \leq 2\log k + 6,
\end{aligned}$$

since $\ell = \lceil \log n \rceil \leq \log k$. □

For $f(k) \leq 2\log k + 6$, we have $\Delta_{f-1}(k) = O(\log^* k)$. Thus, by Lemmas 2.1 and 2.2, we can combine the Leader Election implementation in Section 2.1 with the Group Election implementation in Figure 1, to obtain a Leader Election implementation that has expected step complexity $O(\log^* k)$, from $O(n \log n)$ registers. We can improve the space complexity to $O(n)$ by observing that with probability $1 - 1/n$ only the first $O(\log n)$ `GroupElect` objects are used, and thus we can replace the remaining ones by dummy `GroupElect` objects in which all participating processes get elected.

THEOREM 2.3. *There is an adaptive randomized implementation of a* `LeaderElect` *object from $O(n)$ registers that has expected step complexity $O(\log^* k)$ against the location-oblivious adversary.*

2.3 R/W-Oblivious Adversary

Alistarh and Aspnes [2] presented a randomized implementation of Leader Election with $O(\log \log n)$ expected step complexity against an R/W-oblivious adversary. At the heart of their implementation is a simple Group Election algorithm, which they referred to as *sifting*. Each process participating in this Group Election either writes or reads a shared register. It writes with probability π (which is a parameter) and reads with probability $1 - \pi$, independently of other processes. A process gets elected if and only if it writes, or it reads before any process writes. The first part of the Leader Election implementation in [2] consists of $O(\log \log n)$ rounds of sifting, in which only processes that get elected in a round continue to the next one. The authors show that if the probability parameter π for each round is carefully chosen, then the number of processes that continue after round i, is at most $n^{(1-\epsilon)^i}$, with high probability.

We can use the above Group Election implementation together with the implementation in Section 2.1, to obtain a Leader Election implementation that has expected step complexity $O(\log \log n)$ and uses $O(n)$ registers. This implementation is not adaptive. However we can use a collection of non-adaptive `LeaderElect` objects implemented in that way, to obtain an adaptive implementation, as we sketch now. The main idea is to use $\lceil \log \log \log n \rceil$ such objects $LE_0, LE_1, \ldots$ of increasing size, such that LE_i is for $n_i = 2^{2^{2^i}}$ processes (except for $LE_{\lceil \log \log \log n \rceil}$, which is for n processes). In each LE_i, processes participate only in the first $\Theta(\log \log n_i) = \Theta(2^i)$ Group Elections, and after that, all processes that have not lost and have not won any `Splitter` object yet proceed to the next object LE_{i+1}. The winner of each LE_i participates in a chain of 2-process `LeaderElect` objects which determines the final winner. The expected step complexity of this implementation is $O(\log \log k)$. The intuition is that after $\Theta(\log \log k)$ steps a process ends up in an object LE_i of the 'right' size, such that $\log \log n_i = \Theta(\log \log k)$.

THEOREM 2.4. *There is a randomized implementation of an adaptive* `LeaderElect` *object from $O(n)$ registers that has expected step complexity $O(\log \log k)$ against the R/W-oblivious adversary.*

3. SPACE-EFFICIENT ADAPTIVE ADVERSARY LEADER ELECTION

We describe a Leader Election implementation for the adaptive adversary model, that has step complexity $O(\log k)$ both in expectation and w.h.p. (i.e., with probability

$1 - 1/k^{\Omega(1)}$), and uses $\Theta(n)$ registers. It is a modification of the `RatRace` algorithm proposed by Alistarh et al. [3], which has the same asymptotic step complexity, but uses $\Theta(n^3)$ registers.

3.1 Overview of RatRace

`RatRace` [3] uses two shared memory structures, a *primary tree* and a *backup grid*.

The primary tree is a complete binary tree of height $3 \log n$, where each node v is associated with an `RSplitter` (randomized splitter) object SP_v, and with a randomized 3-process `LeaderElect` object LE_v (implemented from two 2-process `LeaderElect` objects.)

Each process p starts at the root of the primary tree and moves downwards, towards the leaves, until it wins an `RSplitter` object, or it falls off the tree (which happens with low probability). Precisely, when at node v, process p calls SP_v.`split()`. If the call returns L or R then p moves to the left or the right child of v, respectively, provided that v is not a leaf; if v is a leaf, then the process proceeds to the backup grid as we will explain later. If p wins SP_v then it stops moving downwards, and begins to move upwards towards the root, along the same path. At each node v that p visits on its path to the root (including the node at which p won the `RSplitter` object), the process tries to win the `LeaderElect` object LE_v. If p loses LE_v then it immediately loses the implemented `LeaderElect` object; if it wins LE_v then it moves to the parent of v in the tree. The process that wins the `LeaderElect` object at the root competes against the winner of the backup grid.

The backup grid is an $n \times n$ square grid, where each node $v = (i, j) \in \{1, \ldots, n\}^2$ is associated with a (deterministic) `Splitter` object, and a randomized 3-process `LeaderElect` object. By convention, the left and right children of node (i, j) are nodes $(i + 1, j)$ and $(i, j + 1)$, respectively. Each process that falls off the primary tree, starts at node $(1, 1)$ of the grid, and proceeds in the same way as in the primary tree: first it tries to win a `Splitter` object, moving from a node to one of its children when it fails, and then tries to move back to node $(1, 1)$ along the same path, by winning all the `LeaderElect` objects along the way. The properties of deterministic splitters guarantee that the process wins a splitter before it falls off the grid.

Finally, the winners of the `LeaderElect` objects at node $(1, 1)$ of the backup grid and at the root of the primary tree participate in a randomized 2-process `LeaderElect` object, which determines the winner of `RatRace`.

3.2 Improving the Space Complexity

`RatRace` needs $\Theta(2^{3 \log n}) = \Theta(n^3)$ registers for the primary tree of height $3 \log n$, and $\Theta(n^2)$ registers for the backup $n \times n$ grid. To reduce the space complexity we will use the following structure, which we call *elimination path*. It is similar to the backup grid, but uses fewer registers.

An *elimination path of length* ℓ is an ℓ-node path where each node $i \in \{1, \ldots, \ell\}$ is associated with a deterministic `Splitter` object SP_i, and a randomized 2-process `LeaderElect` object LE_i. A process p starts at node $i = 1$, and tries to win SP_i. If p's SP_i.`split()` call returns L then p loses and takes no more steps; if it returns R then p moves to the right, i.e., it moves to node $i + 1$ if $i < \ell$, or it falls off the path if $i = \ell$. Finally, if p wins SP_i then it stops moving to the right, and starts moving to the left towards node 1. From node $i > 1$, it moves to $i - 1$ only if it wins LE_i, otherwise, it loses and stops taking steps. The winner of the elimination path is the process that wins LE_1. The next claim follows from properties of deterministic splitters.

CLAIM 3.1. *If at most ℓ processes enter an elimination path of length ℓ, then no process falls off the right end of the path.*

We replace the backup grid of `RatRace` by an elimination path of length n, which has the same asymptotic step complexity as the backup grid, but uses only $\Theta(n)$ registers.

Further, we replace `RatRace`'s primary tree of height $3 \log n$, by a structure consisting of a smaller primary tree, of height $\log n$, and $n/\log n$ elimination paths EP_i, for $1 \leq i \leq n/\log n$, where each path EP_i has length $4 \log n$. The total number of registers used is $\Theta(2^{\log n} + (4 \log n) \cdot n/\log n) = \Theta(n)$. The smaller primary tree is used in the same way as before, but now any node that falls off enters one of the elimination paths EP_i instead of the backup grid. Precisely, a process that falls off the j-th leaf from the left (of the n leaves in total) moves to the beginning of path $EP_{\lceil j/\log n \rceil}$. The winner of each path EP_i moves back to the primary tree, at leaf i, and from that leaf it tries to reach the root as in the original `RatRace` algorithm. Any process that falls off a path EP_i proceeds to the elimination path of length n that replaced the backup grid. Finally, similarly to `RatRace`, the winner of that path and the winner of the primary tree compete against each other to determine the winner of the implemented `LeaderElect` object.

The step complexity of the implementation for the case of $\log k \leq (\log n)/3$ follows from the analysis of `RatRace`. For the analysis of the complementary case, $\log k > (\log n)/3$, we need the simple claim below, which guarantees w.h.p. that the number of processes that access any given path EP_i is no greater than the length of the path. Thus, by Claim 3.1, we have w.h.p. that no process enters the backup elimination path of length n, which yields an $O(\log n) = O(\log k)$ bound on the step complexity.

CLAIM 3.2. *For any fixed set of $\log n$ leaves, with probability $1 - 1/n^2$ at most $4 \log n$ processes reach those leaves.*

PROOF. The number of processes that reach the $\log n$ leaves is dominated by the number of balls that fall in a fixed set of $\log n$ bins in the classic bins-and-balls model with n balls and n bins, in which every ball is placed in a bin chosen independently and uniformly at random: We can assume that each process p comes with a uniformly random bit-string of length $\log n$. If p tries to win an `RSplitter` object of a node at distance $i - 1$ from the root but fails, then the i-th bit in the bit-string determines whether p's `split()` call returns L or R. Hence, the random bit-string uniquely determines the leaf that p will reach, if it does not win any `RSplitter` object along the way.

The claim then follows by applying standard Chernoff bounds. □

4. ADVERSARY INDEPENDENCE

The Leader Election implementation for the adaptive adversary presented in Section 3 has the same step complexity, $\Theta(\log k)$, even if the adversary is an oblivious one. On the other hand, the implementations in Section 2, which assume a weaker adversary, may need up to $\Theta(k)$ steps in expectation when scheduled by an adaptive adversary. In this

section we describe how we can combine these implementations to obtain one that has the step complexity of RatRace against an adaptive adversary, and the step complexity of the algorithms in Section 2 against a weak adversary.

THEOREM 4.1. *For any Leader Election implementation A for the location-oblivious (resp. R/W-oblivious) adversary, there is a Leader Election implementation that has the same asymptotic step complexity as A against the location-oblivious (resp. R/W-oblivious) adversary, and it has step complexity $O(\log k)$ (both in expectation and w.p. $1-1/k$) against the adaptive adversary. The space complexity of this implementation is $\Theta(n)$ plus the space complexity of A.*

Combining Theorem 4.1 with Theorems 2.3 and 2.4, yields the following.

COROLLARY 4.2. *There is a Leader Election implementation that has an expected step complexity of $O(\log^* k)$ (resp. $O(\log\log k)$) against the location-oblivious (resp. R/W-oblivious) adversary, and a step complexity of $O(\log k)$ (both in expectation and w.p. $1-1/k$) against the adaptive adversary. The space complexity is $\Theta(n)$.*

We now present an implementation that achieves the step and space complexity prescribed in Theorem 4.1. The implementation runs both RatRace and A in parallel, in a round robin fashion. Precisely, each process executes a step of RatRace in every odd step, and a step of A in every even step. A natural way to combine the two interleaved executions would be that each process takes steps until it either wins or loses in one of the two executions; if it loses it also loses in the combined implementation, and if it wins it competes against the winner of the other execution. This approach, however, could yield a combined execution where *no* process wins. For instance, suppose that A is also RatRace. Then it is possible that a process p loses against some process q on one of the LeaderElect objects in the one execution, and at the same time q loses against p on a LeaderElect object in the other execution; thus both processes lose in the combined execution.

To solve this problem we impose the rule that if a process loses in A when it has already won a (deterministic or randomized) splitter object in RatRace, then the process continues to execute RatRace. Precisely, we use the following rules to combine the two executions with the help of an auxiliary LeaderElect object LE_{top}.

1. If a process wins in either RatRace or A then it stops taking steps in the other execution, and it tries to win LE_{top}; if it wins LE_{top} then it wins the implemented LeaderElect object, otherwise it loses.

2. If a process loses in RatRace then it stops taking steps in A, and it loses the implemented LeaderElect object.

3. If a process loses in A *and it has not yet won any of the* Splitter *or* RSplitter *objects in* RatRace, then it stops taking steps in RatRace, and it loses the implemented LeaderElect object.

We sketch the proof of the claim that it is not possible to have a failure-free execution of this combined implementation in which *no* process wins. First we consider the case when no process wins a splitter object in RatRace. Then no processes wins or loses in RatRace, and thus RatRace has no effect on A's execution. Since A is executed without any interference, exactly one process wins A. This process will also win LE_{top}, since no process wins RatRace, and thus no other process participates in LE_{top}.

Now suppose that at least one process p wins a splitter object in RatRace but no process wins the implemented leader election. In particular, p does not win the implemented leader election, so it loses a LeaderElect object in RatRace. If this happens then some other process q wins that LeaderElect object, which implies that q won some splitter object before. Similarly to p, q can lose only if it loses a LeaderElect object in RatRace. Further, if the LeaderElect object that p lost is in a node at distance ℓ from the root, then q can only lose a LeaderElect object at distance at most $\ell-1$ from the root. By iterating this argument, we obtain that some process w wins RatRace. By Rule 1, process w stops A and tries to win LE_{top}. Thus, there will be some process that wins LE_{top} (either w or the winner of A) and thus some process wins the implemented leader election.

Next we sketch the proof of the step complexity of the implementation. First we consider an adaptive adversary. We can view the processes that stop in RatRace because they win or lose in A, as crashed by a (randomized) adaptive adversary. (This adversary runs A in order to decide which processes to stop and when.) Since each process that wins or loses in RatRace stops taking steps in A, it follows that the step complexity of the implementation is bounded asymptotically by the step complexity of RatRace (plus the step complexity of LE_{top}).

We now consider the location-oblivious (resp. R/W-oblivious) adversary. We treat the processes that stop in A because of winning or losing RatRace, as crashed by a (randomized) location-oblivious adversary (resp. R/W oblivious adversary).[1] Unlike in the previous case, now we cannot immediately claim that the step complexity of the implementation is bounded by the step complexity of A, because a process that loses in A may continue to take steps in RatRace. Let t be the maximum number of steps of A that any process takes. Then, t is bounded by the step complexity of A. Moreover, by Rule 3, any process that loses in A but continues to take steps in RatRace must have won a splitter object at some node at distance at most t from the root. Then, from the analysis of RatRace it follows that the extra steps that those processes take are $O(t)$ in expectation and also w.p. $1-1/2^{\Omega(t)}$. It follows that the step complexity of the implementation is asymptotically bounded by that of A.

5. A SPACE LOWER BOUND

We prove a space lower bound of $\Omega(\log n)$ registers for any Leader Election algorithm, and thus for the implementation of TAS objects.

An algorithm satisfies *nondeterministic solo-termination*, if for any configuration and any process p, there is an execu-

[1]For this argument it is important that A works against a location- or R/W-oblivious adversary, rather than just against the adaptive adversary. We cannot view stopped processes as crashed by a randomized oblivious adversary, because the processes stopped and the time at which they stop may depend on random choices of A: these choices affect which processes lose in A and thus which processes stop in RatRace, and this in turn affects which processes lose in RatRace and thus stop in A.

tion in which no process other than p takes any steps, and p finishes its method call within a finite number of steps [10]. Hence, a process is guaranteed to finish its method call with positive probability, whenever there is no interference from other processes. For deterministic algorithms, nondeterministic solo-termination is the same as obstruction-freedom and weaker than wait-freedom.

THEOREM 5.1. *Any nondeterministic solo-terminating Leader Election algorithm requires $\Omega(\log n)$ registers.*

5.1 Proof Overview

We consider a Leader Election algorithm that satisfies nondeterministic solo-termination. Since we prove a space (and not a time) lower bound, we can fix the random decisions made by processes.

The idea is to use a covering argument (as introduced by Burns and Lynch [8]). Suppose n processes run a Leader Election algorithm, where n is a power of two. We let each process take steps until it is poised to write to a register (we say it *covers* that register). Now we schedule processes in rounds. Our goal is to ensure that after the k-th round every register is covered by at most $n-k$ processes. In particular, after $n-4$ rounds every register will be covered by at most 4 processes. When we schedule processes, some processes may see others, and then they can decide to lose their Leader Election. We try to maintain as many "undecided" processes as possible. We will manage to have $\Omega(\log n)$ undecided processes at the end of the $n-4$ rounds, and all of them are covering registers. Since every register is covered by at most 4 processes, the space lower bound follows.

In order to make this work, we have to be careful how to schedule processes in each round. We maintain the invariant that after the k-th round no process has ever written to a register that is covered by fewer than $n-k$ processes. Now consider some set $R = \{r_1, \ldots, r_\ell\}$ of registers, such that each register r_j, $1 \le j \le \ell$, is covered by exactly $n-k$ processes. For each register r_j we choose one process q_j that covers r_j, and let it write to r_j. This way, all registers that stored any useful information will get overwritten by processes $q_1, \ldots, q_\ell$. Since registers in $\overline{R}$ have not yet been written at all in the entire past execution, processes $q_1, \ldots, q_\ell$ will not gain any additional information (except about themselves) when only *they* take additional steps. It follows that if we let these processes run solo, one after the other, then one of them, say q_1, must win. Since all registers in R are covered by other processes, q_1 must write to a register not in R before it can win. We stop q_1 when it is poised to write for the first time to a register that is not in R. This way, we now have increased the number of processes covering registers outside of R by one and at the same time every register in R is only covered by $n-k-1$ processes. We may have reduced the number of "undecided" processes, because $q_2, \ldots, q_\ell$ may have lost their Leader Election already.

Note that if not all of them have lost their Leader Election, then the assumption that no process knows about any other process that has not yet lost might not strictly be true anymore. We deal with this problem by grouping all processes that know about each other together, and we count always only one representative from each group when we determine the number of processes that cover a register. Whenever the representative overwrites a register at the beginning of the round, and then proceeds to take steps, we let its entire group also take steps. This way, we can maintain the invariant that one of the processes in such a group writes outside of R.

5.2 Proof of the Space Lower Bound

This section is devoted to the proof of Theorem 5.1.

We let $\mathcal{P}$ denote a set of n processes in the system, and $\mathcal{R}$ the set of registers of the system. We assume w.l.o.g. that n is a power of two. A *configuration* C is a tuple $(s_1, \ldots, s_n, v_1, v_2, \ldots, v_{|\mathcal{R}|})$, denoting that the i-th process is in state s_i, and the j-th register has value v_j. Configurations will be denoted by capital letters, and the initial configuration is denoted C_{init}. Two configurations $C = (s_1, \ldots, s_n, v_1, \ldots, v_m)$ and $C' = (s'_1, \ldots, s'_n, v'_1, \ldots, v'_m)$ are *indistinguishable* to the i-th process in $\mathcal{P}$, if $s_i = s'_i$ and $v_j = v'_j$ for all $j \in \{1, \ldots, m\}$. Configurations C and C' are indistinguishable to a set Q of processes, if they are indistinguishable to every process $q \in Q$.

A *schedule* σ is a (possibly infinite) sequence of processes. An *execution* $E(C, \sigma)$ is a sequence of steps beginning in configuration C and moving through successive configurations one at a time. At each step, the next process p_i indicated in the schedule σ, takes the next step in its program. Since our computation model is nondeterministic, we fix the nondeterministic decisions made by processes in our lower bound proof. We use an arbitrary (but fixed) one that guarantees that each process p_i terminates within a bounded number of steps if it runs solo. If σ is a finite schedule, the final configuration of the execution $E(C, \sigma)$ is denoted $\sigma(C)$ or $C\big(E(C, \sigma)\big)$. If σ and π are finite schedules then $\sigma\pi$ denotes the concatenation of σ and π; similarly, we can concatenate execution by letting $E(C, \sigma) \cdot E\big(\sigma(C), \pi\big) = E(C, \sigma\pi)$. Let Q be a set of processes, and σ a schedule. If only processes in Q appear in σ, then σ is a *Q-only* schedule and $E(C, \sigma)$ is a *Q-only* execution.

We say that in configuration C a process p *covers* a register r, if the state of p as determined by C is such that in its next step p will write register r. We assume w.l.o.g. that whenever a process writes a value to a register, that value is a pair (x, ID), where ID is the process' identifier. Initially, the second component of every pair stored in a register is $\perp$. We say process q is *visible* on register r in some configuration, if r's value is (x, q) for some x. (Thus, initially no process is visible on any register.) We say process p *sees* process q in some step of an execution, if in that step p reads a register on which q is visible. Every execution E defines a relation "$\leftrightarrow_E$" over the set $\mathcal{P}$ of processes, where $p \leftrightarrow_E q$, if $p = q$ or during E either p sees q or q sees p. Then $\leftrightarrow_E$ is reflexive and symmetric. Let $\equiv_E$ be the transitive closure of $\leftrightarrow_E$, thus $\equiv_E$ is an equivalence relation over $\mathcal{P}$.

CLAIM 5.2. *Let E be some execution starting in the initial configuration C_{init}, and $Q \subseteq \mathcal{P}$ the set of processes that finish their Leader Election algorithm during E. If $Q \neq \emptyset$ is closed w.r.t. $\equiv_E$, then exactly one process in Q wins during E.*

PROOF. Remove from E all steps by processes in $\overline{Q}$ and denote D the resulting execution. Then configurations $C(E)$ and $C(D)$ are indistinguishable to all processes in Q. It follows that during D all processes in Q finish their Leader Election with the same result as in E. Since only processes from Q take steps during D, exactly one of them wins. □

CLAIM 5.3. *Let $R = \{r_1, \ldots, r_\ell\} \subseteq \mathcal{R}$, $R' = \{r_{\ell+1}, \ldots, r_v\} \subseteq \mathcal{R} - R$, and let E be an execution starting*

in C_{init}, such that in $C(E)$ no process is visible on any register in $\overline{R}$. Further, let $Q \subseteq \mathcal{P}$ be a set that is closed w.r.t. $\equiv_E$. Suppose in $C = C(E)$, every register $r_j \in R \cup R'$, $1 \leq j \leq v$, is covered by some process $p_j \in \overline{Q}$, and every register $r_i \in R$, $1 \leq i \leq \ell$, by some process $q_i \in Q$. Then there exists a Q-only schedule β such that during $E(C, \beta)$ at least one process writes to a register in $\overline{R \cup R'}$, and no process in Q sees a process in $\overline{Q}$.

PROOF. Let σ be the Q-only schedule where each process q_j, $1 \leq j \leq \ell$, takes exactly one step, and σ' the Q-only schedule such that during the execution $E(C, \sigma\sigma')$ every process in Q finishes its algorithm. Nondeterministic solo-termination guarantees that such a schedule σ' exists. Let $\beta = \sigma\sigma'$ and $E' = E(C, \beta)$. For the purpose of a contradiction, suppose that during E' no process writes to a register in $\overline{R \cup R'}$.

During E', all register contents of registers in R are overwritten by processes in Q. Thus, throughout E', no process in $\overline{Q}$ is ever visible on any register, and thus no process in Q sees a process in $\overline{Q}$. Since in addition processes in $\overline{Q}$ take no steps during E', set Q is closed w.r.t. $\equiv_{E'}$. By assumption, it is also closed w.r.t. $\equiv_E$, and thus w.r.t. $\equiv_{E \cdot E'}$.

By the assumption that no process writes to a register in $\overline{R \cup R'}$ during execution E', in configuration $C' = C(E')$, no process is visible on a register in $\overline{R \cup R'}$. Consider a $\overline{Q}$-only execution E'' that starts in configuration C', and where first all processes $p_1, \ldots, p_v$ take exactly one step, and then we let all processes that haven't finished their algorithm, yet, run to completion. Since the algorithm satisfies nondeterministic solo-termination, such an execution E'' exists. Since in E'' processes $p_1, \ldots, p_v$ first overwrite all registers in $R \cup R'$, no process in $\overline{Q}$ sees any process in Q during E''. Trivially, no process in Q sees any process in $\overline{Q}$. Thus, Q is closed w.r.t. $\equiv_{E''}$. Since Q is closed w.r.t. $\equiv_{E \cdot E'}$, it is also closed w.r.t. $\equiv_{E \cdot E' \cdot E''}$. This implies that $\overline{Q}$ is also closed w.r.t. $\equiv_{E \cdot E' \cdot E''}$. Since all processes in Q and in $\overline{Q}$ finish their algorithm during $E \cdot E' \cdot E''$, we conclude from Claim 5.2 that some process in Q and some process in $\overline{Q}$ wins—a contradiction. □

Recall that we assumed that n is a power of two.

LEMMA 5.4. *For every $k \in \{0, \ldots, n-1\}$, there exists*

- *a schedule α_k, defining execution $E_k = E(C_{init}, \alpha_k)$ and configuration $C_k = C(E_k)$;*
- *a partition $(Q_1, \ldots, Q_{m_k})$ of $\mathcal{P}$; and*
- *m_k processes $q_j \in Q_j$, $1 \leq j \leq m_k$;*

such that in configuration C_k

(a) every process q_j, $1 \leq j \leq m_k$, covers some register;

(b) no register is covered by more than $n-k$ processes in $\{q_1, \ldots, q_{m_k}\}$;

(c) if a process is visible on a register r, then r is covered by $n-k$ processes in $\{q_1, \ldots, q_{m_k}\}$;

(d) each set Q_j, $1 \leq j \leq m_k$, is closed w.r.t. $\equiv_{E_k}$; and

(e) $m_0 = n$ and $m_k \geq m_{k-1} - \lfloor m_{k-1}/(n-k+1) \rfloor + 1$ for $k > 0$.

PROOF. We prove the lemma by induction on k.

First consider the base case, $k = 0$. If we let a process p run solo when no process is visible on any register, p must win the Leader Election (by Claim 5.2). This requires that p writes to at least one register, or else some other process will win the Leader Election in a solo-run that follows p's. Hence, we can let each process run solo as long as it doesn't write to a register and stop the process when it is poised to write to a register for the first time. Thus, eventually all processes are poised to write, while at the same time no process is visible on any register. Let α_0 be the schedule that results in such an execution $E_0 = E(C_{\text{init}}, \alpha_0)$, where no process writes and where in $C_0 = C(E_0)$ every process is poised to write to a register. Further, let $m_0 = n$ and q_i the i-th process in $\mathcal{P}$ and $Q_j = \{q_j\}$ for $1 \leq j \leq n$. In C_0 all processes cover some register, no register is covered by more than $n - 0$ processes, no process is visible on any register, the sets $Q_1, \ldots, Q_n$ are equivalence classes of $\equiv_{E_0}$, and $m_0 = n$. Hence, the induction hypothesis is true for $k = 0$.

Now assume the hypothesis is true for some integer $k \in \{0, \ldots, n-2\}$. Let α_k, m_k, $Q_1, \ldots, Q_{m_k}$ and $q_1, \ldots, q_{m_k}$ be given such that (a)-(e) are satisfied. Let $R = \{r_1, \ldots, r_\ell\}$ be the set of registers that are covered by exactly $n-k$ processes in $\{q_1, \ldots, q_{m_k}\}$ in configuration C_k. If $R = \emptyset$, we let $\alpha_{k+1} = \alpha_k$, and the claim for $k' = k+1$ follows immediately from the induction hypothesis for k.

Now suppose that $R \neq \emptyset$. Choose some indices $i_1, \ldots, i_\ell \in \{1, \ldots, m_k\}$ such that process q_{i_j}, $1 \leq j \leq \ell$, covers register r_j and let $Q = Q_{i_1} \cup \cdots \cup Q_{i_\ell}$. Assume w.l.o.g. that $(i_1, \ldots, i_\ell) = (m_k - \ell + 1, \ldots, m_k)$. The set Q is closed w.r.t. $\equiv_{E_k}$ because all sets Q_j, $m_k - \ell + 1 \leq j \leq m_k$, are closed w.r.t. $\equiv_{E_k}$.

Now let R' be the set of registers covered by exactly $n-k-1$ processes in $\{q_1, \ldots, q_{m_k}\}$. By definition, no process in Q covers a register in R', and since $n - k - 1 \geq 1$, every register in R' is covered by at least one process in $\overline{Q}$. Moreover, since $n - k \geq 2$, every register in R is covered by exactly two processes in $\{q_1, \ldots, q_{m_k}\}$ out of which exactly one is in Q. Hence, every register in R is covered by one process in Q and by at least one process in $\overline{Q}$. Finally, by induction hypothesis (c), in C_k no process is visible on any register in $\overline{R}$.

Thus, we can apply Claim 5.3. It follows that there exists a Q-only schedule β_k such that during $E(C_k, \beta_k)$ at least one process writes to a register in $\overline{R \cup R'}$, and no process in Q sees a process in $\overline{Q}$. We let β'_k be the longest prefix of β_k such that in $E(C_k, \beta'_k)$ no process writes to a register in $\overline{R \cup R'}$. Thus, at the end of $E(C_k, \beta'_k)$ some process $q \in Q$ is poised to write to a register in $\overline{R \cup R'}$. We let $\alpha_{k+1} = \alpha_k \beta_k$, $E_{k+1} = E_k \cdot E(C_k, \beta'_k) = E(C_{\text{init}}, \alpha_{k+1})$, and $C_{k+1} = C(E_{k+1})$. Moreover, let $m_{k+1} = m_k - \ell + 1$, $Q_{m_{k+1}} = Q$, $q'_{m_{k+1}} = q$, and for $1 \leq j < m_{k+1}$ let $Q'_j = Q_j$ and $q'_j = q_j$.

Then processes in $Q'_1 \cup \cdots \cup Q'_{m_{k+1}-1}$ take no steps during $E(C_k, \beta'_k)$, so in C_{k+1} every process q'_j, $1 \leq j < m_{k+1}$ covers some register. By construction, at the end of execution $E(C_k, \beta'_k)$ process $q'_{m_{k+1}}$ is poised to write, so it also covers some register. This proves (a).

A process in $\{q'_1, \ldots, q'_{m_{k+1}-1}\}$ takes no step during $E(C_k, \beta'_k)$, so in configuration C_{k+1} it covers exactly the same register as in configuration C_k. Now consider some register r. If $r \in R$, then in configuration C_k register r is covered by exactly $n-k$ processes in $\{q_1, \ldots, q_{m_k}\}$. Ex-

actly one of these $n-k$ processes is in $\{q_{m_k-\ell+1},\dots,q_{m_k}\}$, and thus not in $\{q'_1,\dots,q'_{m_{k+1}-1}\}$. Thus, exactly $n-k-1$ processes in $\{q'_1,\dots,q'_{m_{k+1}-1}\}$ cover register r in configuration C_{k+1}. If $r \in R'$, then in configuration C_k register r is covered by exactly $n-k-1$ processes in $\{q_1,\dots,q_{m_k}\}$, none of whom takes a step in $E(C_k,\beta'_k)$. Hence, in C_{k+1} every register $r \in R \cup R'$ is covered by exactly $n-k-1$ processes in $\{q'_1,\dots,q'_{m_{k+1}-1}\}$. Moreover, in C_{k+1} process $q'_{m_{k+1}}$ covers a register from $\overline{R \cup R'}$, so $r \in R \cup R'$ is covered by exactly $n-k-1$ processes in $\{q'_1,\dots,q'_{m_{k+1}}\}$. Now suppose $r \in \overline{R \cup R'}$, so in C_k register r is covered by at most $n-k-2$ processes in $\{q_1,\dots,q_{m_k}\}$. By construction of β'_k, during $E(C_k,\beta'_k)$ exactly one processes, namely $q'_{m_{k+1}}$, becomes poised to write to a register in $\overline{R \cup R'}$. Hence, in C_{k+1} register $r \in \overline{R \cup R'}$ is covered by at most $n-k-1$ processes in $\{q_1,\dots,q_{m_k}\}$, and thus also by at most $n-k-1$ processes in $\{q'_1,\dots,q'_{m_{k+1}}\}$. This proves (b).

Now consider a register r on which some process is visible in configuration C_{k+1}. In configuration C_k, no process is visible on a register in $\overline{R}$, and in execution $E(C_k,\beta'_k)$ no process writes to a register in $\overline{R \cup R'}$. Hence, $r \in R \cup R'$. As argued above in the proof of part (b), in C_{k+1}, every register in $R \cup R'$ is covered by exactly $n-k-1$ processes in $\{q'_1,\dots,q'_{m_{k+1}-1}\}$. This proves (c).

By construction and Claim 5.3, no process in Q sees any process in $\overline{Q}$ during execution $E(C_k,\beta'_k)$. Since that execution is Q-only, no process in $\overline{Q}$ sees any process during that execution. It follows that each of the sets $Q'_1 = Q_1,\dots,Q'_{m_{k+1}-1} = Q_{m_{k+1}-1}$ (which is are subsets of $\overline{Q}$) is closed w.r.t. $\equiv_{E(C_k,\beta'_k)}$. By the induction hypothesis each of those sets is closed w.r.t. $\equiv_{E_k}$, so they are also closed w.r.t. $\equiv_{E_{k+1}}$. Similarly, $Q = Q'_{m_{k+1}}$ is closed w.r.t. $\equiv_{E(C_k,\beta'_k)}$, and since Q is the union of sets which are closed w.r.t. $\equiv_{E_k}$, Q is closed w.r.t. $\equiv_{E_{k+1}}$. This proves (d).

By construction, in configuration C_k every register in R is covered by at least $n-k$ processes from $\{q_1,\dots,q_{m_k}\}$, so $|R| \le \lfloor m_k/(n-k) \rfloor$. Moreover,

$$m_{k+1} = m_k - |R| + 1 \ge m_k - \left\lfloor \frac{m_k}{n-k} \right\rfloor + 1.$$

This proves (e). □

Define

$$f(0) = n \quad \text{and}$$
$$f(k+1) = f(k) - \left\lfloor \frac{f(k)}{n-k} \right\rfloor + 1 \quad \text{for } k \ge 0.$$

Further, let for $k \ge 1$

$$\delta(k+1) = f(k) - f(k+1) = \left\lfloor \frac{f(k)}{n-k} \right\rfloor - 1.$$

Claim 5.5. *For an integer $s \ge 0$ and $k \in I(s) := \left\{n - \frac{n}{2^s},\dots,n - \frac{n}{2^{s+1}} - 1\right\}$,*

(a) $f(k) = n \cdot \frac{s+1}{2^s} - s \cdot \left(k - n + \frac{n}{2^s}\right)$, and

(b) $\delta(k+1) = s$.

Proof. We first argue that statement (a) implies statement (b). Let $d = k - n + n/2^s$ and suppose (a) is true. Then

$$\begin{aligned}\delta(k+1)+1 &= \left\lfloor \frac{f(k)}{n-k} \right\rfloor = \left\lfloor \frac{n\frac{s+1}{2^s} - s\cdot\left(k-n+\frac{n}{2^s}\right)}{n-k} \right\rfloor \\ &= \left\lfloor \frac{n\frac{s+1}{2^s} - s\cdot d}{n-(d+n-n/2^s)} \right\rfloor = \left\lfloor \frac{n\frac{s+1}{2^s} - s\cdot d}{n/2^s - d} \right\rfloor \\ &= \left\lfloor \frac{s+1-s\cdot d\cdot 2^s/n}{1-d\cdot 2^s/n} \right\rfloor = \left\lfloor \frac{s(1-d\cdot 2^s/n)+1}{1-d\cdot 2^s/n} \right\rfloor \\ &= \left\lfloor s + \frac{1}{1-d\cdot 2^s/n} \right\rfloor = s + \lfloor \zeta \rfloor, \quad (1)\end{aligned}$$

where

$$\zeta = \frac{1}{1-d\cdot 2^s/n} = \frac{n}{n-d\cdot 2^s}.$$

Since $n - n/2^s \le k \le n - n/2^{s+1} - 1$, we have

$$0 \le d \le n - n/2^{s+1} - 1 - n + n/2^s = n/2^{s+1} - 1, \quad (2)$$

and thus

$$1 \le \zeta \le \frac{n}{n-(n/2^{s+1}-1)\cdot 2^s} = \frac{n}{n/2+2^s} < 2.$$

Thus, $\lfloor \zeta \rfloor = 1$, so from (1) we obtain $\delta(k+1)+1 = s+1$, which proves (b).

We now prove statement (a) by induction on k. If $k=0$, then $k \in I(0)$ and (a) is true. Thus, suppose that (a) and thus also (b) hold for some value of k. Then

$$\begin{aligned}f(k+1) = f(k) - \delta(k+1) &= n\frac{s+1}{2^s} - s\left(k - n + \frac{n}{2^s}\right) - s \\ &= n\frac{s+1}{2^s} - s\left(k+1-n+\frac{n}{2^s}\right). \quad (3)\end{aligned}$$

If $k < n - n/2^{s+1} - 1$, then $k+1 \in I(s)$ and the claim is proven. Now suppose $k = n - n/2^{s+1} - 1$, i.e., $k+1 \in I(s+1)$. Then from (3) we get

$$\begin{aligned}f(k+1) &= n\cdot\frac{s+1}{2^s} - s\left(n - \frac{n}{2^{s+1}} - n + \frac{n}{2^s}\right) \\ &= n\cdot\frac{2s+2}{2^{s+1}} - s\cdot\frac{n}{2^{s+1}} = n\cdot\frac{s+2}{2^{s+1}}.\end{aligned}$$

This proves (a). □

Proof of Theorem 5.1. We can assume w.l.o.g. that n is a power of two. Also, since we want to prove a space lower bound, we can fix arbitrary random choices by the algorithm, and thus the algorithm becomes deterministic and obstruction free.

Let $k = n-4$. By Lemma 5.4, there exists an execution E_k that starts in configuration C_{init} and ends in configuration C_k, such that in C_k at least m_k processes cover registers, but no register is covered by more than $n-k=4$ processes. This implies that there must be at least $\lceil m_k/4 \rceil$ registers.

Note that from part (e) of Lemma 5.4 and the definition of f, we immediately get $m_k \ge f(k)$. Since $k = n-4$, we have $k = n - n/2^{\log n - 2}$. By the definition of $I(s)$ from Claim 5.5, $k \in I(\log n - 2)$. Thus, by that same claim,

$$\begin{aligned}m_{n-4} &\ge f(n-4) \\ &= n\cdot\frac{\log n - 1}{n/4} - (\log n - 2)\left(n - 4 - n + \frac{n}{n/4}\right) \\ &= 4(\log n - 1).\end{aligned}$$

Hence, at least $\log n - 1$ registers are covered in configuration C_{n-4}. □

6. A 2-PROCESS TIME LOWER BOUND

THEOREM 6.1. *For any randomized TAS implementation that can be accessed by two processes, and any integer $t > 0$, there is a schedule (determined by an oblivious adversary) such that with probability at least $1/4^t$ some process does not finish its* `TAS()` *method within fewer than t steps.*

PROOF. The proof is by Yao's Min-Max Lemma [14]. Let $\mathcal{A}_t$ denote the set of all *deterministic* TAS algorithms for two processes in which no process takes more than t steps. Under the (standard) assumption that the domain of values for each register is countable, we have that the set $\mathcal{A}_t$ is also countable.[2] Let $\mathcal{S}_t$ denote the set of all possible schedules (i.e., sequences of process IDs) for two processes, in which each process appears exactly t times. Then,

$$|\mathcal{S}_t| = \binom{2t}{t} \leq 2^{2t}. \tag{4}$$

For any $A \in \mathcal{A}_t$ and $S \in \mathcal{S}_t$, we denote by $c(A, S)$ the indicator function that is 1 if and only if at least one of the two process in algorithm A takes t steps under schedule S. Let R be any randomized TAS implementation for two processes, and let R_t be the same algorithm except that processes are stopped after they execute their t-th step (if they have not finished before that step). We can view R_t as a probability distribution over the countable set $\mathcal{A}_t$. The probability that at least one process takes t or more steps in R under schedule S is then equal to $\mathbf{E}[c(R_t, S)]$, where $c(R_t, S)$ is now a 0/1 random variable depending on the random choices of the randomized algorithm R_t. Therefore, the probability that at least one process takes t or more steps in R for *some* schedule is

$$\max_{S \in \mathcal{S}_t} \mathbf{E}[c(R_t, S)].$$

By Yao's Min-Max Lemma, for any probability distribution D_t over the (finite) set of schedules $\mathcal{S}_t$,

$$\max_{S \in \mathcal{S}_t} \mathbf{E}[c(R_t, S)] \geq \min_{A \in \mathcal{A}_t} \mathbf{E}[c(A, D_t)].$$

We observe that for any $A \in \mathcal{A}_t$ we have $c(A, S) = 1$ for at least one $S \in \mathcal{S}_t$ because of the impossibility of deterministic wait-free implementations for TAS. Thus, by choosing D_t to be the *uniform* distribution over $\mathcal{S}_t$, we obtain that for any $A \in \mathcal{A}_t$, $\mathbf{E}[c(A, D_t)] \geq 1/|\mathcal{S}_t|$. Therefore, by (4) $\max_{S \in \mathcal{S}_t} \mathbf{E}[c(R_t, S)] \geq 1/|\mathcal{S}_t| \geq 1/2^{2t}$. And since this is true for an arbitrary randomized TAS implementation R for two processes the theorem follows. □

Conclusion

In this paper we devised several improved randomized TAS algorithms. Most importantly, we have shown that the randomized expected step complexity of TAS is $O(\log^* k)$ (where k is the contention) against the oblivious and some slightly stronger adversary models. The progress in improving randomized TAS algorithms is mirrored by recent progress on randomized consensus algorithms. Just this year, Aspnes [5] devised a randomized binary consensus algorithm that has $O(\log \log n)$ expected step complexity against an oblivious adversary. This algorithm is based on the sifting technique from [2]. It would be interesting to know whether techniques similar to those presented here can be used to find further improvements on the randomized complexity of consensus.

Several other important problems remain open. For example even for the oblivious adversary it is still not known, whether a TAS implementation with constant expected step complexity exist. No non-trivial lower bounds on the expected step complexity of Leader Election are known, not even in the adaptive adversary model. Finally, there is still an exponential gap between the lower bound of $\Omega(\log n)$ and the upper bound of $O(n)$ for the number of registers that are required for randomized or obstruction-free TAS algorithms.

[2]This assumption allows us to apply Yao's Min-Max Lemma in a straight-forward way. However, with a slightly more technical argument, a variant of the Min-Max Lemma can be used even if $\mathcal{A}_t$ is not countable.

7. REFERENCES

[1] Y. Afek, E. Gafni, J. Tromp, and P. M. B. Vitányi. Wait-free test-and-set. In *Proc. of 6th WDAG*, pages 85–94, 1992.

[2] D. Alistarh and J. Aspnes. Sub-logarithmic test-and-set against a weak adversary. In *Proc. of 25th DISC*, pages 97–109, 2011.

[3] D. Alistarh, H. Attiya, S. Gilbert, A. Giurgiu, and R. Guerraoui. Fast randomized test-and-set and renaming. In *Proc. of 24th DISC*, pages 94–108, 2010.

[4] J. Aspnes. A modular approach to shared-memory consensus, with applications to the probabilistic-write model. In *Proc. of 20th PODC*, pages 460–467, 2010.

[5] J. Aspnes. Faster randomized consensus with an oblivious adversary. In *Proc. of 31st PODC*, 2012. To appear.

[6] H. Attiya and K. Censor-Hillel. Lower bounds for randomized consensus under a weak adversary. *SIAM J. on Comp.*, 39(8):3885–3904, 2010.

[7] H. Attiya, F. Kuhn, C. G. Plaxton, M. Wattenhofer, and R. Wattenhofer. Efficient adaptive collect using randomization. *Distr. Comp.*, 18(3):179–188, 2006.

[8] J. E. Burns and N. A. Lynch. Bounds on shared memory for mutual exclusion. *Inf. Comput.*, 107(2):171–184, 1993.

[9] W. Eberly, L. Higham, and J. Warpechowska-Gruca. Long-lived, fast, waitfree renaming with optimal name space and high throughput. In *Proc. of 12th DISC*, pages 149–160, 1998.

[10] F. E. Fich, M. Herlihy, and N. Shavit. On the space complexity of randomized synchronization. *J. of the ACM*, 45(5):843–862, 1998.

[11] W. Golab, D. Hendler, and P. Woelfel. An $o(1)$ RMRs leader election algorithm. *SIAM J. on Comp.*, 39:2726–2760, 2010.

[12] M. Moir and J. H. Anderson. Fast, long-lived renaming. In *Proc. of 8th WDAG*, pages 141–155, 1994.

[13] J. Tromp and P. M. B. Vitányi. Randomized two-process wait-free test-and-set. *Distr. Comp.*, 15(3):127–135, 2002.

[14] A. C.-C. Yao. Probabilistic computations: Towards a unified measure of complexity. In *Proc. of 17th FOCS*, pages 222–227, 1977.