Randomized loose renaming in O(log log n) time

Dan Alistarh, James Aspnes, George Giakkoupis, Philipp Woelfel

▶ **To cite this version:**

Dan Alistarh, James Aspnes, George Giakkoupis, Philipp Woelfel. Randomized loose renaming in O(log log n) time. 32nd ACM Symposium on Principles of Distributed Computing (PODC), Jul 2013, Montreal, Canada. hal-00856744

HAL Id: hal-00856744
https://inria.hal.science/hal-00856744

Submitted on 2 Sep 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Randomized Loose Renaming in O(log log *n*) Time

[Extended Abstract]

Dan Alistarh*
Computer Science and Artificial Intelligence Lab, MIT
alistarh@mit.edu

James Aspnes†
Dept. of Computer Science
Yale University
aspnes@cs.yale.edu

George Giakkoupis‡
INRIA Rennes – Bretagne Atlantique
george.giakkoupis@inria.fr

Philipp Woelfel§
Dept. of Computer Science
University of Calgary
woelfel@ucalgary.ca

ABSTRACT

Renaming is a classic distributed coordination task in which a set of processes must pick distinct identifiers from a small namespace. In this paper, we consider the time complexity of this problem when the namespace is linear in the number of participants, a variant known as loose renaming. We give a non-adaptive algorithm with $O(\log\log n)$ (individual) step complexity, where n is a known upper bound on contention, and an adaptive algorithm with step complexity $O((\log\log k)^2)$, where k is the actual contention in the execution. We also present a variant of the adaptive algorithm which requires $O(k \log\log k)$ *total* process steps. All upper bounds hold with high probability against a strong adaptive adversary.

We complement the algorithms with an $\Omega(\log\log n)$ expected time lower bound on the complexity of randomized renaming using test-and-set operations and linear space. The result is based on a new coupling technique, and is the first to apply to non-adaptive randomized renaming. Since our algorithms use $O(n)$ test-and-set objects, our results provide matching bounds on the cost of loose renaming in this setting.

*This author was supported by the SNF Postdoctoral Fellows Program, NSF grant CCF-1217921, DoE ASCR grant ER26116/DE-SC0008923, and by grants from the Oracle and Intel corporations.

†Supported in part by NSF grant CCF-0916389.

‡This work was funded in part by INRIA Associate Team RADCON, and ERC Starting Grant GOSSPLE 204742.

§This research was undertaken, in part, thanks to funding from the Canada Research Chairs program and the HP Labs Innovation Research Program.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PODC'13, July 22–24, 2013, Montréal, Québec, Canada.
Copyright 2013 ACM 978-1-4503-2065-8/13/07 ...$15.00.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Distributed Data Structures; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

Keywords

Distributed computing, shared memory, renaming, upper bounds, lower bounds

1. INTRODUCTION

The *renaming* problem [8] is a fundamental task in distributed computing. It can be seen as the dual of *consensus* [33]: if to reach consensus processes must *agree* on a single value, for renaming processes must *disagree* constructively by returning distinct identifiers from a small namespace. Considerable effort, e.g., [8, 9, 13, 17], went into analyzing the feasibility and complexity of renaming in asynchronous shared-memory and message-passing systems. From a theoretical perspective, the problem is known to have a rich structure, in particular given its connections to algebraic topology, e.g., [18, 19, 28]. On the other hand, renaming is known to be related to practical problems such as mutual exclusion [5], counting [4], or concurrent memory management [27].

A significant amount of research, e.g., [2, 10, 31], studied efficient renaming in asynchronous shared memory. Early work focused on *non-adaptive* renaming, where the maximum number of processes n is known, and each process must obtain a unique name from a target namespace of size m, where m is a function of n. If $m = n$, i.e., the namespace size is optimal, then renaming is *strong* (or *tight*); otherwise, renaming is *loose*. A stronger variant of the problem is *adaptive* renaming [1], where the size of the target namespace and the complexity of the algorithm must depend on the contention k in the current execution, as opposed to the number of processes in the system, n.

Randomization has proved a very useful tool for getting fast renaming algorithms. Intuitively, a process can simply pick a name at random, repeating the choice in case of a collision. If the space is large enough, then the expected number of collisions is small. Using a similar idea, Panconesi, Papatriantafilou, Tsigas and Vitányi [32] obtained

a loose renaming algorithm with poly-logarithmic expected step complexity against a strong adversary. Further work, e.g., [4,6,20], resulted in algorithms for strong renaming and with logarithmic expected step complexity. Recently, [5] gave a lower bound of $\Omega(\log(k/c))$ expected process steps for adaptive renaming into a namespace of size ck, for any constant $c \geq 1$, extending an information-based technique of Jayanti [29]. This result suggested that the logarithmic complexity threshold is inherent for adaptive renaming, and that no asymptotic complexity gain can be obtained by relaxing the namespace size within constant factors.

In this paper, we contradict this intuition by presenting two renaming algorithms which achieve linear namespace size with *sub-logarithmic* step complexity. Our first algorithm, called `ReBatching`, uses $(1+\epsilon)n$ names for any fixed constant $\epsilon > 0$, and all processes finish it in time $O(\log\log n)$, with high probability. The second algorithm is adaptive, and all processes obtain names of value $O(k)$ in $O((\log\log k)^2)$ steps, both with high probability, where k is the contention in the execution. We also give a more complex variant of this second algorithm with $O(\log\log k)$ *average* step complexity. The algorithms use test-and-set (`TAS`) operations and linear space.

Both our algorithms circumvent the logarithmic lower bound of [5], but in different ways. `ReBatching` is not affected by the bound since it is not adaptive, while the adaptive algorithm circumvents it since the ck namespace bound is ensured only with high probability, rather than with probability 1.

On the negative side, we prove a lower bound of $\Omega(\log\log n)$ expected worst-case steps for renaming algorithms which use only `TAS` primitives and linear space in n. Since both our algorithms verify these assumptions, they are time-optimal in this setting.

Our algorithms work against a strong adaptive adversary, which can examine the processes' entire state when deciding on scheduling and crashes. They improve exponentially on the best previously known algorithms [6, 32]. Our lower bound is the first to apply to non-adaptive randomized renaming, and exhibits a new trade-off between space and expected running time for this problem.

The intuition behind our algorithms is simple: processes share a sequence of indexed shared memory locations; a process obtains a name by performing a successful `TAS` on a location, returning the index of that location as its name. If unsuccessful, the process tries again. Thus, the key to obtaining a fast algorithm is to minimize the contention between process probes.

In this context, `ReBatching` allocates a set of $2n$ locations, split into disjoint batches $B_0, B_1, \ldots$ of decreasing length, such that batch B_i has approximately $n/2^i$ consecutive locations. Each process will perform a *constant* number of probes in each batch, until it first acquires a location. The key idea in the analysis is that, as the execution progresses to later batches, the number of processes surviving up to some batch B_ℓ is proportional to $O(n/2^{2^\ell})$, while the space available in the batch is $\Theta(n/2^\ell)$. This phenomenon perpetuates so that, when $\ell = \log\log n$, there are essentially no more processes competing.

The adaptive algorithm works as follows. Processes share a set of `ReBatching` objects $R_1, R_2, \ldots$, where object number i provides a distinct namespace of size $\Theta(2^i)$. Processes first "race" to obtain a unique name by accessing objects R_{2^ℓ} for $\ell = 0, 1, \ldots$, until successful in some object. Since the names obtained may be super-linear in k, processes proceed to a second *search* phase, in which they "crunch" the namespace by essentially running by binary search on the `ReBatching` objects. The step complexity of the algorithm is $O((\log\log k)^2)$ and all processes obtain names of value $O(k)$, both with high probability. We also consider a more complex version, in which processes pipeline their steps in the `ReBatching` objects to amortize the complexity of the failed `ReBatching` calls. The resulting algorithm has total step complexity $O(k\log\log k)$ with high probability, and the same namespace guarantees as the simpler version.

Our lower bound considers algorithms using linear space and `TAS` operations, and proves that any such algorithm must cause some process to take $\Theta(\log\log n)$ steps with constant probability. We first reduce the problem to one where, to obtain a name, a process must *win* a `TAS` (i.e., change the value of that location). Even given this reduction, processes do learn new information from their unsuccessful probes, which can lead to complex correlations of future probes. To circumvent this issue, we construct an execution in which processes are carefully pruned in each round to ensure independence between the survivors. This reduces the problem to one where each process loses each trial with a fixed probability, independently of the actions of other processes. In turn, this will show that some processes must still take steps after $\Theta(\log\log n)$ rounds.

The main technical ingredient of the lower bound is a coupling gadget which guarantees that the number of processes accessing and leaving a location is a Poisson random variable, ensuring independence. The resulting worst-case execution is composed of layers in which each process takes steps in randomly permuted order. Such an execution can in fact be created by an *oblivious* adversary, so the lower bound works in this weaker adversarial model as well. Interestingly, both our algorithms match this time bound, and work even for an adaptive adversary.

Our work can be seen as part of a wider effort investigating sub-logarithmic-time algorithms for fundamental distributed tasks. Tight bounds of $\Theta(\log n/\log\log n)$ are known for randomized mutual exclusion against a strong adversary [23, 25, 26]. Against a weak adversary, randomized algorithms with $O(\log\log n)$ complexity have been recently given for test-and-set [3, 22] and consensus [7], while Bender and Gilbert [14] provide a mutual exclusion algorithm with $O((\log\log n)^2)$ amortized expected RMR complexity. However, prior to our paper, no algorithms with $O(\log\log n)$ step complexity were known against a strong adversary for *any* non-trivial problem in asynchronous shared-memory.

2. MODEL AND PROBLEM STATEMENT

We assume the standard asynchronous shared memory model with n processes $p_1, \ldots, p_n$. Processes follow an algorithm, composed of *steps*. Without loss of generality, each step is comprised of local computations, and one shared memory step. Processes share registers, on which they can perform `TAS` operations. We say that a process *wins* a `TAS` if it successfully changes the value of the register and returns 0; otherwise, the process *loses* the `TAS`, returning 1. Any number of processes may fail by crashing. A failed process does not take further steps in the execution.

We consider randomized algorithms, in which processes' actions may depend on the outcomes of local random coin

flips. We assume that each process starts with an initial name from an unbounded namespace, and that processes share a consistent indexing of the memory locations on which they perform shared-memory operations.

The order in which processes take steps and their crashes are controlled by an *adversary*. In this paper, we consider two standard types of adversarial schedulers. The *adaptive* (or *strong*) adversary is allowed to see the state of all processes (including the results of coin flips) when making its scheduling choices. The weaker *oblivious* adversary knows the algorithm, but not the results of coin flips when deciding the schedule.

The renaming problem [8] is defined as follows. Given a target namespace size $m \geq n$, each of the n processes must eventually return a unique name v_i between 1 and m. A correct algorithm must guarantee *termination*, *namespace size* and *uniqueness* in every execution. In the classic (non-adaptive) renaming problem [8], the parameters n and m are known by the processes.

The *adaptive* renaming problem [1], which we also consider in this paper, requires that the complexity of the protocol and the size of the resulting namespace should only depend on the number of participating processes k in the current execution. In this paper, we also relax the *namespace size* requirement to be probabilistic.

We focus on two complexity measures. The first is *(individual) step complexity*, i.e., the maximum number of steps that *any* process performs in an execution. The second measure is *total step complexity* (also known as *work*) which counts the *total* number of steps that *all* processes perform during the execution.

We say that an event occurs *with high probability (w.h.p.)* if its probability is at least $1 - 1/n^c$, where $c > 1$ is a constant. For adaptive algorithms, where the bound on n may not be known, we will express high probability as $1 - 1/k^c$, where k is the contention in the execution.

Test-and-Set vs. Read-Write. Previous work on this problem, e.g., [4,6,32], considered the read-write shared-memory model, implemented randomized `TAS` out of reads and writes, and then solved renaming on top of `TAS`. In this paper, we assume hardware `TAS` is given. Otherwise, we could implement randomized adaptive `TAS` which we could then use as part of our algorithms. This would increase our expected worst-case complexity by a multiplicative $O(\log \log k)$ factor.[1] On the other hand, if we only employ read-write registers, the high probability bounds for our algorithms become at least logarithmic, as logarithmic w.h.p. bounds are inherent even for two-process randomized `TAS` out of reads and writes [22].

3. RELATED WORK

Renaming was introduced in [8], and early work focused on its solvability in asynchronous crash-prone settings [8,13,17], showing that $(2n-1)$-renaming can be achieved in message-passing and read-write shared-memory. A namespace lower bound of $(2n-1)$ was shown by Herlihy and Shavit [28], highlighting deep connections with algebraic topology. Castañeda and Rajsbaum [18, 19] further characterized the namespace size, while some of the results were recently re-derived by Attiya and Paz [11] using counting arguments. Significant effort went into obtaining efficient deterministic algorithms, e.g., [2, 10, 31]. We refer the reader to [16] for a survey of deterministic and long-lived solutions.

Panconesi et al. [32] were the first to use randomization, and gave an algorithm guaranteeing a namespace of size $(1+\epsilon)n$, with $O(M \log^2 n)$ expected running time, where M is the size of the initial namespace, and $\epsilon > 0$ is a constant. This cost can be reduced to $O(\text{polylog}\, n)$ if adaptive test-and-set [6, 22] is used. Eberly, Higham, and Warpechowska-Gruca [20] obtained strong long-lived randomized renaming with amortized step complexity $O(n \log n)$. Reference [6] gave an algorithm guaranteeing a namespace of size ck and running in time $O(\log^2 k)$, both with high probability in k. In [4], Alistarh, Aspnes, Censor-Hillel, Gilbert and Zadimoghaddam gave a strong adaptive algorithm with $O(\log k)$ step complexity, which is optimal for these namespace requirements [5]. All these references implement test-and-set out of read-write registers, while we assume that test-and-set is given in hardware. We discuss our results in the read-write model at the end of Section 2.

Reference [5] shows a linear time lower bound for deterministic implementations of renaming in a polynomial namespace in n, and a logarithmic lower bound on the expected step complexity of adaptive ck-renaming against a strong adversary, when the namespace size is guaranteed in every execution. Both our algorithms circumvent the second bound, as discussed in Section 1.

The idea of splitting the available space into several disjoint levels to minimize the number of collisions, used in the `ReBatching` algorithm, is similar to the multi-level hashing schemes by Broder and Karlin [15] and Fotakis, Pagh, Sanders and Spirakis [21]. However, we consider a concurrent setting with an adaptive adversary, and use different analysis techniques.

4. NON-ADAPTIVE ALGORITHM

We present the `ReBatching` algorithm (for *relaxed batching*), which solves renaming using a namespace of size $(1+\epsilon)n$, and has step complexity $O(\log \log n)$ w.h.p.

The algorithm allocates a shared memory array of `TAS` objects of size $m = (1+\epsilon)n$, for some constant $\epsilon > 0$ that is a parameter of the algorithm. The idea is that each shared `TAS` object is associated with a name, and a process needs to win the `TAS` operation in order to acquire that name. The key to the performance of the algorithm is to probe `TAS` objects in a random fashion, but in a way that minimizes the number of failed probes by each process. E.g., if processes do just uniform random probes among all objects, then with probability $1 - o(1)$ some process will have to do $\Omega(\log n)$ probes before it acquires a name.

The `TAS` objects are arranged into disjoint *batches* $B_0, \ldots, B_\kappa$, where $\kappa = \lceil \log \log n \rceil$, and $|B_i| = b_i$ with

$$b_i = \begin{cases} n, & \text{if } i = 0; \\ \lceil \epsilon n / 2^i \rceil, & \text{if } 1 \leq i \leq \kappa. \end{cases} \quad (1)$$

We assume that n is sufficiently large so that the total size of batches does not exceed $m = (1+\epsilon)n$. (We have $\sum_i b_i \leq n + \sum_{1 \leq i \leq \kappa} (\epsilon n / 2^i + 1) = (1+\epsilon)n - \epsilon n / 2^\kappa + \kappa$, and $\epsilon n / 2^\kappa$ is greater than $\kappa = \lceil \log \log n \rceil$ when n is greater than a sufficiently large constant.)

[1] This holds since each `TAS` is accessed by $O(\log k)$ processes in our algorithm, w.h.p. Also notice that the linearization issues pointed out in [24] are circumvented since we only require simple leader election algorithms to make our algorithms work, as opposed to fully linearizable `TAS` objects.

Class ReBatching(n, ϵ)

/* $m = \lceil (1+\epsilon)n \rceil$ */
shared: array $B[0 \dots m-1]$ of TAS objects
/* **for each** $0 \le i \le \kappa = \lceil \log\log n \rceil$, $B_i = B[s_i..s_i + b_i - 1]$, **where** $s_i = \sum_{0 \le j < i} b_j$ **and** b_i **is given in** (1) */

Method GetName()

1 **for** $0 \le i \le \kappa$ **do**
2 | $u \leftarrow$ TryGetName(i)
3 | **if** $u \neq -1$ **return** u
4 **end**
/* backup phase */
5 **for** $0 \le u \le m-1$ **do**
6 | **if** $B[u]$.TAS() $= 0$ **return** u
7 **end**
8 **return** -1

Method TryGetName(i)

/* t_i **is defined in** (2) */
9 **for** $1 \le j \le t_i$ **do**
10 | choose $x \in \{0, \dots, b_i - 1\}$ unif. at random
11 | **if** $B_i[x]$.TAS() $= 0$ **return** $s_i + x$
12 **end**
13 **return** -1

Figure 1: The ReBatching **algorithm.**

To acquire a name, a process accesses the batches in increasing order of their index i. For each batch B_i, the process calls TAS on (at most) t_i objects from that batch chosen independently and uniformly at random. The process stops as soon as it wins one TAS operation. The number t_i of probes by a process on batch B_i is

$$t_i = \begin{cases} \lceil 17 \ln(8e/\epsilon)/\epsilon \rceil, & \text{if } i = 0; \\ 1, & \text{if } 1 \le i \le \kappa - 1; \\ \beta, & \text{if } i = \kappa, \end{cases} \tag{2}$$

where $\beta \ge 1$ is a constant that can be tuned to achieve the desired high probability on the event that all processes acquire names by these probes. As a backup, processes that fail to win a TAS despite trying on all batches proceed to call TAS on *all* objects sequentially. Our analysis will show that this backup phase is executed with very low probability.

Pseudocode for the algorithm is given in Figure 1.

Analysis. The proof of correctness for the algorithm is straightforward, therefore we focus on the analysis of its running time. In the rest of this section we prove the following theorem.

THEOREM 4.1. *For any fixed $\epsilon > 0$, the* ReBatching *algorithm for a namespace of size $(1+\epsilon)n$ uses $O(n)$* TAS *objects and achieves w.h.p. step complexity at most $\log\log n + O(1)$ and total step complexity $O(n)$ against an adaptive adversary. (Both bounds hold also in expectation.)*

For each $1 \le i \le \kappa + 1$, let n_i be the total number of processes that execute t_{i-1} probes on B_{i-1} but fail in all of them to acquire a name. I.e., their call TryGetName$(i-1)$ returns -1 (line 2 of the pseudocode), and thus they must then call TryGetName(i) if $i \le \kappa$, or run the backup phase if $i = \kappa + 1$. We will show now that n_i drops roughly as $n/2^{2^i}$. Let

$$n_i^* = \begin{cases} \epsilon n / 2^{2^i + i + \delta}, & \text{if } 1 \le i \le \kappa - 1; \\ \log^2 n, & \text{if } i = \kappa, \end{cases}$$

where $\delta > 0$ is an arbitrary small constant; let also $n_{\kappa+1}^* = 0$. Recall that β is the number t_κ of probes on B_κ.

LEMMA 4.2. *With probability $1 - 1/n^{\beta - o(1)}$, we have that $n_i \le n_i^*$ for all $1 \le i \le \kappa + 1$.*

PROOF. We will bound the probability of the events

$$\mathcal{E}_i := \begin{cases} n_1 > n_1^*, & \text{if } i = 0; \\ (n_{i+1} > n_{i+1}^*) \wedge (n_i \le n_i^*), & \text{if } 1 \le i \le \kappa. \end{cases}$$

The lemma the follows by an application of the union bound.

We bound first the probability that $n_1 \ge n_1^*$, i.e., at least n_1^* processes fail in all their t_0 probes on B_0. We assume that for each of the n process, all the t_0 random choices of objects from B_0 that the process will probe are decided in advance (and revealed to the adaptive adversary). Of course the process may end up not probing all those objects, as it may win some TAS sooner or crash. Then, the event that n_1^* of the n processes fail in all their t_0 probes on B_0, occurs only if the following event occurs: There is a set P of n_1^* processes, and a set $L \subseteq B_0$ of n_1^* objects from B_0, such that no process from P chooses any object from L. The reason is that since the size of B_0 is $b_0 = n$, for each process that fails to win any TAS in B_0 there is a distinct object in B_0 that is not probed by any process. For a fixed pair of sets P and L, the probability that no process from P chooses any object from L is

$$\left(1 - \frac{|L|}{b_0}\right)^{t_0 |P|} = \left(1 - \frac{n_1^*}{n}\right)^{t_0 n_1^*} \le e^{-t_0 (n_1^*)^2/n} = e^{-t_0 \alpha^2 n},$$

where $\alpha := n_1^*/n = \epsilon / 2^{3+\delta}$. Further, the number of possibilities to choose a set P of n_1^* out of n processes and a set L of n_1^* objects from B_0 is

$$\binom{n}{n_1^*}\binom{b_0}{n_1^*} = \binom{n}{n_1^*}^2 \le \left(\frac{en}{n_1^*}\right)^{2n_1^*} = \left(\frac{e}{\alpha}\right)^{2\alpha n} = e^{2\alpha n \ln(e/\alpha)}.$$

From the union bound then, the probability that there is at least one pair P, L such that no process from set P chooses any object from set L is bounded by the product $e^{-t_0\alpha^2 n} \cdot e^{2\alpha n \ln(e/\alpha)} = e^{-\Omega(n)}$. Therefore, the same bound holds for the event that $n_1 \ge n_1^*$, and thus

$$\Pr(\mathcal{E}_0) \le \Pr(n_1 \ge n_1^*) = e^{-\Omega(n)}.$$

Next we bound the probability of event $\mathcal{E}_i = (n_i \le n_i^*) \wedge (n_{i+1} > n_{i+1}^*)$, for the case of $1 \le i \le \kappa - 1$. Recall that each process does at most $t_i = 1$ probe on B_i. Consider all processes that do a probe on B_i but fail to win the TAS operation, and let $p_1, p_2, \dots$ be the list of those processes, in the order in which they finish their last TAS operation on B_{i-1}. W.l.o.g. we assume that a list $\ell_1, \ell_2, \dots$ of objects from B_i is chosen in advanced, each object chosen independently and uniformly at random, and that process p_j probes object ℓ_j of B_i if it is scheduled to do such a probe. We can now relate n_{i+1} to the number of collisions in list $\ell_1, \ell_2, \dots$; a *collision* occurs in position j if $\ell_j = \ell_{j'}$ for some $j' < j$. The total number of collisions in the first n_i positions is

then an upper bound on the number n_{i+1} of processes that do an unsuccessful probe in B_i. Since the event $\mathcal{E}_i$ we are interested in holds only if $n_i \leq n_i^*$, we consider just the first n_i^* entries in the list, $\ell_1, \ldots, \ell_{n_i^*}$. The probability of a collision in position j is at most $(j-1)/b_i$, as at most $j-1$ out of the b_i objects in B_i have already been selected. Thus, if X_j is the indicator random variable of the event that there is a collision in position j, then

$$\Pr(X_j = 1 \mid \ell_1 \ldots \ell_{j-1}) \leq (j-1)/b_i. \tag{3}$$

The expectation of the total number of collisions in the first n_i^* positions is then

$$\mathbf{E}\left[\sum_{j=1}^{n_i^*} X_j\right] \leq \sum_{j=1}^{n_i^*} \frac{j-1}{b_i} \leq \frac{(n_i^*)^2/2}{b_i} \leq \frac{(\epsilon n/2^{2^i+i+\delta})^2/2}{\epsilon n/2^i}$$
$$= \epsilon n/2^{2^{i+1}+i+1+2\delta} \leq n_{i+1}^*/2^\delta.$$

Further, because of the special type of dependence (3) between the X_j, a simple coupling argument gives that the above sum of X_j is dominated by the sum of n_i^* *independent* binary random variables Y_j with $\Pr(Y_j) = (j-1)/b_i$ (see e.g., [12, Lemma 3.1], for a similar result). It follows then from Chernoff bounds that

$$\Pr\left(\sum_{j=1}^{n_i^*} X_j > n_{i+1}^*\right) = e^{-\Omega(n_{i+1}^*)} = 1/n^{\omega(1)},$$

and this implies $\Pr(\mathcal{E}_i) = 1/n^{\omega(1)}$.

It remains to bound the probability of $\mathcal{E}_\kappa = (n_\kappa \leq n_\kappa^*) \wedge (n_{\kappa+1} > 0)$. As before we decide in advance the $t_\kappa = \beta$ random choices of objects from B_i to be probed by the j-th process that does an unsuccessful probe on $B_{\kappa-1}$; let L_j denote the multi-set of those choices. It suffice to consider just $L_1, \ldots, L_{n_\kappa^*}$, as $\mathcal{E}_\kappa$ occurs only if $n_\kappa \leq n_\kappa^*$. We have that $n_{\kappa+1} > 0$ holds only if there is some $j \leq n_\kappa^*$ for which $L_j \subseteq \bigcup_{j \neq j' \leq n_\kappa^*} L_{j'}$, i.e., for some j, each of the objects contained in L_j is contained also in at least one other $L_{j'}$. As $\left|\bigcup_{j \neq j' \leq n_\kappa^*} L_{j'}\right| \leq t_\kappa(n_\kappa^* - 1)$, the probability of the latter event is bounded by

$$n_\kappa^* \left(\frac{t_\kappa(n_\kappa^* - 1)}{b_\kappa}\right)^{t_\kappa} = 1/n^{\beta - o(1)},$$

and thus $\Pr(\mathcal{E}_\kappa) = 1/n^{\beta-o(1)}$.

To complete the proof of Lemma 4.2, we observe that the event we are interested in, that for all $1 \leq i \leq \kappa+1$ it holds $n_i \leq n_i^*$, is equivalent to $\bigwedge_{i=0}^{\kappa} \overline{\mathcal{E}_i} = 1 - \bigvee_{i=0}^{\kappa} \mathcal{E}_i$. The lemma now follows from the bounds we have shown for $\Pr(\mathcal{E}_i)$ and the union bound. □

From Lemma 4.2 it follows that with probability $1 - 1/n^{\beta-o(1)}$, no process reaches the backup phase, any process executes at most $t_0 + (\kappa-1)\cdot 1 + \beta = \log\log n + O(1)$ probes on all batches, the total number of probes by all processes is at most $nt_0 + \sum_{1 \leq i \leq \kappa} n_i^* t_i = O(n)$. Therefore, w.h.p. the step complexity is at most $\log\log n + O(1)$ and the total step complexity $O(n)$. Further, each process executes $O(n)$ steps in the worst-case (i.e., when it has to enter the backup phase and probe all m locations), thus the worst-case step complexity is $O(n)$ and the worst-case total step complexity is $O(n^2)$. It follows that for $\beta \geq 2$ the expected step complexity is $\log\log n + O(1)$, and for $\beta \geq 3$ the expected total step complexity is $O(n)$. This completes the proof of Theorem 4.1.

5. ADAPTIVE ALGORITHMS

Next we present two adaptive renaming algorithms. The first one has step complexity $O((\log\log k)^2)$ w.h.p., and the second has total step complexity $O(k \log\log k)$ w.h.p. In both algorithms, the largest name assigned to any process is $O(k)$ w.h.p.

Our algorithms don't need to know the number n of processes in the system, but if they don't then they require unbounded space. In fact, for ease of exposition we present our algorithms in such a way that they use an unbounded number of `ReBatching` objects $R_1, R_2, \ldots$, where R_i provides a namespace of size $O(2^i)$ and thus is constructed from $O(2^i)$ `TAS` objects. If n is known, it is straight forward to modify the algorithms so that they use only the first $2^{\log n+1}$ `TAS` objects and thus $O(n)$ `TAS` objects in total are sufficient.

5.1 Adaptive ReBatching

We describe now the algorithm with step complexity $O((\log\log k)^2)$, which we call `AdaptiveReBatching`. The algorithm uses slightly modified `ReBatching` objects, in which the backup phase (lines 5–7 in Figure 1) is omitted, and thus a `GetName` call returns -1 if no name is acquired during any of the `TryGetName` calls (in line 2). A collection $R_1, R_2, \ldots$ of such `ReBatching` objects is used, where object R_i provides a namespace of size $m_i = (1+\epsilon)n_i$ with $n_i = 2^i$. Precisely, the namespace of R_i is $\{s_i, \ldots, s_i + m_i - 1\}$, where $s_i = \sum_{1 \leq j < i} m_j$.

Each process p first tries to get a name by doubling the index of the `ReBatching` object it accesses after each unsuccessful trial. Precisely, p repeatedly calls R_{2^ℓ}.`GetName` for $\ell = 0, 1, \ldots$ until it succeeds in getting a name; let 2^{ℓ^*} be the index of the object from which p gets that name. Next, p tries to acquire a smaller name by doing a binary search on objects $R_{2^{\ell^*-1}+1}, \ldots, R_{2^{\ell^*}}$. Precisely, p initially sets $a \leftarrow 2^{\ell^*-1}+1$ and $b \leftarrow 2^{\ell^*}$; while $a < b$, it sets $d \leftarrow \lfloor (a+b)/2 \rfloor$, and calls R_d.`GetName`; if the call returns a name then the value of b is updated setting $b \leftarrow d$, otherwise a's value is updated letting $a \leftarrow d+1$. Once $a \geq b$, p stops and gets assigned the name it acquired from R_b.

THEOREM 5.1. *The* `AdaptiveReBatching` *algorithm has step complexity $O((\log\log k)^2)$ w.h.p., and the largest name it assigns to any process is $O(k)$ w.h.p.*

PROOF. From Theorem 4.1 and Lemma 4.2, we obtain that w.h.p. all `GetName` calls to objects R_i with index $i \geq \log k$ (and thus $n_i \geq k$) succeed in returning a name and complete in $O(\log\log n_i)$ steps. Precisely, for each $i \geq \log k$, all R_i.`GetName` calls succeed with probability $1 - 1/n_i^c$, for a constant $c > 0$, and thus by the union bound, all R_i.`GetName` calls for *all* $i \geq \log k$ succeed with probability at least $1 - \sum_{i \geq \log k} 1/n_i^c = 1 - O(1/k^c)$.

It follows that in the first part of the algorithm (in which processes access objects R_{2^ℓ} for $\ell = 1, 2, \ldots$), w.h.p. every process acquires a name from some object R_i with $i \leq 2^{\lceil \log\log k \rceil}$, after accessing at most $\lceil \log\log k \rceil + 1$ objects (namely, objects R_ℓ for $0 \leq \ell \leq \lceil \log\log k \rceil$), spending at most $\log\log k + O(1)$ steps on each of them. Thus, processes complete this part in $O((\log\log k)^2)$ steps w.h.p.

In the binary search part, a process searches among at most $2^{\lceil \log\log k \rceil}$ objects, thus it accesses at most $\log\log k +$

$O(1)$ of them, spending again at most $\log\log k + O(1)$ steps on each. Thus, this second part takes $O((\log\log k)^2)$ steps w.h.p., as well.

Finally, since w.h.p. all calls R_i.`GetName` for $i \geq k$ return a name, binary search guarantees that every process will finally obtain a name from some object R_i with $n_i \leq 2^{\lceil \log k \rceil}$. Thus the largest name is w.h.p. at most $\sum_{0 \leq i \leq \lceil \log k \rceil} m_i \leq 4(1+\epsilon)k$. □

5.2 Faster Adaptive ReBatching

The `AdaptiveReBatching` algorithm presented in the previous section has total step complexity $\Theta(k(\log\log k)^2)$. We propose now a variant of this algorithm, called `FastAdaptiveReBatching`, which reduces the total step complexity to $O(k \log\log k)$. Pseudocode for this algorithm is provided in Figure 2.

As before, we use a collection $R_1, R_2, \ldots$ of `ReBatching` objects, where object R_i, for $i \geq 1$, has parameter $n_i = 2^i$, but now we require that $\epsilon = 1$ and thus the namespace of R_i has size $2n_i = 2^{i+1}$. The general idea is the same as in the previous algorithm: A process searches for the smallest index i^*, such that it can acquire a name from R_{i^*} but not from R_{i^*-1} (implying that at least $\Omega(n_{i^*})$ processes participate). The difference to the previous algorithm is that when a process tries to get a name from object R_i, it executes only a constant number of probes on this object, by calling `TryGetName`, as opposed to $\Theta(\log\log n_i)$ many, when calling `GetName`. This may yield "false negative" results, so a process may have to revisit an object R_i again at a later point if it has not already obtained a name from some object R_j with index $j < i$. Therefore, a process keeps track of a lower bound a, and an upper bound b on i^*, as well as the total number t of times it has executed `TryGetName` on object R_a. The upper bound b is "hard" in the sense that the process has already obtained a name u from R_b. The lower bound a is "weak" meaning that it might still be possible for the process to find a name from some object R_i with $i < a$.

Processes try to find i^* and to acquire a name from R_{i^*} using a recursive method `Search`(a, b, u, t). A call to this method requires that $a < b$, that u is a name the process has already acquired from R_b, and that the process has previously called R_a.`TryGetName`(j) for $j = 0, \ldots, t-1$. The method guarantees to return a name u' from some object R_i, $a \leq i \leq b$, that the process has acquired (possibly $u' = u$ and $i = b$). Moreover, if $i > a$, then the process has called `TryGetName`(j) on R_{i-1} for each $j = 0, \ldots, \kappa(i-1)$, where $\kappa(s) = \lceil \log(s) \rceil$ is the maximum batch index in R_s; thus, w.h.p. the number of processes participating is $\Omega(n_i)$, and so u' is not "too large". But if $i = a$, then no guarantees are provided by `Search` on the number of times the process has tried to find a name in R_{i-1}.

`Search`(a, b, u, t) is implemented as follows. If $t > \kappa(a)$, the process can simply return u (in line 11) because it has already executed enough `TryGetName` calls on R_a (the implementation guarantees that $b = a + 1$ in this case). If $t \leq \kappa(a)$, the process executes R_a.`TryGetName`(t), and if the call returns a name, the `Search` method can simply return that name (lines 12–13); in this case a can be used as a new upper bound. Otherwise, the process tries to improve the upper bound b: It chooses the median $d = \lceil (a+b)/2 \rceil$ of the indices $a+1, \ldots, b$ (line 14). If $d < b$, then the process uses d as its new lower bound and tries to obtain a new name u from $R_d, \ldots, R_b$ using a recursive call `Search`$(d, b, u, 0)$ (line 15). If that method returns a name from R_i for some $i > d$, then the ongoing `Search` can finish and return the name u in line 17: It is guaranteed that $\kappa(i-1)$ unsuccessful `TryGetName` calls on R_{i-1} have been performed. If the recursive `Search` call in line 15 returns a name from R_d, then d becomes the new upper bound and the process continues its recursive search by calling `Search`$(a, d, u, t+1)$ in line 16. Finally, in case $d = b$, and thus $b = a + 1$, the process simply calls `Search`$(a, b, u, t+1)$ in line 16, trying to either get a name from R_a, or confirmation that a is the right lower bound.

Class `FastAdaptiveReBatching`

shared: $R_1, R_2, \ldots$, where R_i is a `ReBatching`(n_i, ϵ) object with $n_i = 2^i$ and $\epsilon = 1$

`/* ` R_i`'s namespace is` $\{2^{i+1}, \ldots, 2^{i+2} - 1\}$`. */`

Method `GetName()`

```
1  ℓ ← −1
2  repeat
3  |  ℓ ← ℓ + 1
4  |  u ← R_{2^ℓ}.TryGetName(0)
5  until u ≠ −1
6  while ℓ ≥ 1 and u ∈ R_{2^ℓ} do
7  |  u ← Search(2^{ℓ−1}, 2^ℓ, u, 1)
8  |  ℓ ← ℓ − 1
9  end
10 return u
```

Method `Search`(a, b, u, t)

```
   /* κ(i) = ⌈log(i)⌉                                   */
11 if t > κ(a) return u
12 u' ← R_a.TryGetName(t)
13 if u' ≠ −1 return u'
14 d ← ⌈(a + b)/2⌉
15 if d < b then u ← Search(d, b, u, 0)
16 if u ∈ R_d then u ← Search(a, d, u, t + 1)
17 return u
```

Figure 2: The `FastAdaptiveReBatching` algorithm.

Using `Search` the renaming algorithm works as follows: A process first searches for an initial upper bound similarly to the previous algorithm, by executing `TryGetName(0)` on R_{2^ℓ} for each $\ell = 0, 1, \ldots$, until it finds a name (lines 1–5). Once it has found the first name u in R_{2^ℓ}, its upper bound is $a = 2^\ell$ and its lower bound is $b = 2^{\ell-1}$. Then the process calls the method `Search`$(2^{\ell-1}, 2^\ell, u, 1)$ in line 7. (The last parameter is 1 instead of 0, because $R_2^{\ell-1}$ has already been accessed once by the process.) If this call returns a name in R_i for some $i > 2^{\ell-1}$, then the process can finish and use that name—since it has executed R_{i-1}.`TryGetName`(j) calls for $j = 1, \ldots, \kappa(i-1)$, we have w.h.p. that $\Omega(n_i)$ processes are participating. If the `Search` call returns a name in $R_{2^{\ell-1}}$, then the process has to consider smaller `ReBatching` objects, so it sets $\ell = \ell - 1$ and repeats the `Search` step with the smaller upper and lower bounds.

In the rest of this section we sketch a proof of the following theorem.

THEOREM 5.2. *The* `FastAdaptiveReBatching` *algorithm has total step complexity $O(k \log\log k)$ w.h.p., and the largest name it assigns to any process is $O(k)$ w.h.p.*

To facilitate the proof, we give an equivalent description of the algorithm in terms of the underlying binary search tree on objects $R_1, R_2, \ldots$ In this tree, for each index i that is a power of two, objects $R_1, \ldots, R_{i-1}$ form a perfect binary subtree, and each internal node R_j has the property that objects in its left (right) subtree have indices smaller (resp. greater) than j. (Thus, odd-indexed objects are the leaves, and each internal node's index is the average of its two children's indices.) In the following we will often say 'node i' instead of 'node R_i'. The `FastAdaptiveReBatching` algorithm can now be described in terms of the above tree as follows.

A process p starts from the leftmost leaf (node 1), and walks upwards (along the path $2^0, 2^1, \ldots$), calling `TryGetName(0)` on each node it visits, until it gets a name (lines 1–5). We will see that w.h.p. this happens after traversing at most $\log\log k + O(1)$ nodes, thus p acquires a name from some node $2^\ell = O(\log k)$. Next, p tries to get a smaller name by searching in the left subtree of node 2^ℓ (lines 6–9). In this search, p may visit the same node more than once, and each time it does it calls `TryGetName(`t`)`, where t is the number of times it has visited the node before. Precisely, p visits first the root $2^{\ell-1}$ of the left subtree of node 2^ℓ, and for each node a that p visits:

- If the `TryGetName` call on node a (line 12) fails to return a name, then p proceeds to visit the right child d of a if a is an internal node (line 15); or if a is a leaf, p visits a again (line 16) until it finally gets a name (line 13) or has tried unsuccessfully on all batches of a (line 11).
- Suppose now that the `TryGetName` call on a (line 12) succeeds in acquiring a name. Standard binary search would move to the left child of a, or finish if a is a leaf. Here, however, p tries again on the most recently visited node a' from which p has not succeeded in acquiring a name yet (line 7 or 16). (Node a' can be found by following the upward path from a; a' is the first node to be reached through its right child.) If p succeeds in getting a name from a', then it repeats the above procedure with a' in place of a. Otherwise, it visits the left child of a; or if a is a leaf it keeps trying on a' until it gets a name or fails on all batches of a', as before.

The formal proof that the above description is equivalent to the `FastAdaptiveReBatching` algorithm is omitted due to space restrictions.

We first bound the index of the object from which a process gets a name in the first phase of the algorithm.

Claim 5.3. *W.h.p. in the loop in lines 1–5 every process gets a name from some object R_i with index $i \le i_{max}$, where*

$$i_{max} := 2^{\lceil \log\log k \rceil + 2} < 8 \log k.$$

The claim holds because $n_{i_{max}} \ge k^4$, thus the probability that two fixed processes that access $R_{i_{max}}$ make the same random choice for their first probe is at most $1/n^4$. Taking the union bound over the at most k^2 pairs yields the claim.

Next we bound the total number of steps by processes before they obtain a *small* name, where a name is *small* if it comes from an object R_i with index $i \le i_{min}$ for

$$i_{min} := \lceil \log k \rceil + 2.$$

For each index $i \ge i_{min}$ and batch j of R_i, we bound the number of processes accessing that batch. For simplicity, we assume that the number of probes per process on each batch is the same for all batches, and equal to $t_* := \max_j\{t_j\}$ (see Eq. (2)); thus $t_* = O(1)$.

Let $\mathcal{P}$ denote the path in the tree between nodes i_{max} and i_{min}. For each $i_{min} \le i \le i_{max}$, let h_i be the distance of node i from path $\mathcal{P}$. Further, for each $l \ge 0$, let $k_l = k/2^{2^{l+l-1}}$.

Claim 5.4. *W.h.p. for all pairs i, j with $i_{min} \le i \le i_{max}$ and $0 \le j \le \lceil \log(i) \rceil$, we have that at most k_{h_i+j} processes call R_i.`TryGetName(`j`)`.*

The proof of Claim 5.4 is similar to that of Lemma 4.2, and relies on the following result.

Claim 5.5. *Let $i \ge i_{min}$, $0 \le j \le \lceil \log(i) \rceil$, and $l \ge j$. The probability that at most k_l processes call R_i.`TryGetName(`j`)` and more than k_{l+1} of these calls fail to return a name is bounded by $1/n_i^{t_* - o(1)}$.*

To prove Claim 5.5 we distinguish two cases. If $k_{l+1} = \omega(i)$, then we use the same collision-counting argument as in the proof of Lemma 4.2 (for batches 1 up to $\kappa - 1$). If $k_{l+1} = O(i)$, then we employ the argument used for the last batch in the proof of Lemma 4.2.

To show Claim 5.4, we use Claim 5.5 and the union bound to obtain w.h.p. for all pairs i, j with $i_{min} \le i \le i_{max}$ and $0 \le j \le \lceil \log(i) \rceil$, that either more than k_{h_i+j} processes call R_i.`TryGetName(`j`)` or at most k_{h_i+l+1} of these calls fail to return a name. We complete the proof by showing that if the above event holds, then for every pair i, j at most k_{h_i+l} processes call R_i.`TryGetName(`j`)`. The proof of the last statement is by induction on $l = h_i + j$. In this induction, the more interesting case is when $j = 0$ and $h_i > 0$, and thus we must argue that no more than k_{h_i} processes call R_i.`TryGetName(0)`: Consider the first node $r < i$ in the path P_i from i to $\mathcal{P}$; r the first node along P_i reached through its right child. From $\mathcal{P}$'s definition it follows that such a node r exists and $r \ge i_{min}$. The distance from i to r is $h_i - h_r$, and thus there are $h_i - h_r - 1$ nodes between them. Each node that accesses R_i must have previously successfully obtained names from those $h_i - h_r - 1$ nodes, and must have failed to get names from batches $0, \ldots, (h_i - h_r - 1)$ of R_r. Thus, the number of nodes that access i is bounded by the number of processes that failed to obtain a name by call R_r.`TryGetName(`$h_i - h_r - 1$`)`. The latter number is bounded by $k_{h_r+(h_i-h_r)} = k_{h_i}$, by the induction hypothesis and the event we assumed at the beginning.

From Claim 5.4 it follows that the number of steps by processes on objects $i \ge i_{min}$ is bound w.h.p. by the sum of all k_{h_i+j}; we show this to be $O(k \log\log k)$. Further we show a deterministic bound of $O(k \log\log k)$ on the steps by processes on objects R_i with $i < i_{min}$, before these processes acquire a small name. Thus, we have the following result.

Lemma 5.6. *The total number of steps by all processes before they acquire a small name is $O(k \log\log k)$ w.h.p.*

It remains to bound the steps by processes *after* they have acquired a small name. As processes do not know k, they continue to search for even smaller names. We observe that no process does more than $O(\log\log k)$ *consecutive* failed `TryGetName` calls (on the same or different objects): After the first $O(\log\log k)$ of them the process reaches a leaf, and after $O(\log\log k)$ additional ones the process stops. Further, no more than $O(k)$ `TryGetName` calls can be successful on objects R_i with $i \le i_{min}$, as they have $O(k)$ names in total. The next (deterministic) bound then follows.

Lemma 5.7. *The total number of steps by processes after they acquire a small name is bounded by $O(k \log\log k)$.*

To complete the proof of Theorem 5.2, we argue that w.h.p. every process acquires a name from some object R_i with $i \leq \lceil \log k \rceil$. The reason is that a process returns a name from R_i only after it has tried and failed on all batches of R_{i-1}, and if $i > \lceil \log k \rceil$ this happens with probability polynomially small in k.

6. LOWER BOUND

Our lower bound shows that, under reasonable conditions, an oblivious adversary can force some process in any loose renaming algorithm using only TAS objects to take $\Omega(\log \log n)$ steps. For simplicity, the lower bound assumes a non-adaptive algorithm. Formally, we show:

THEOREM 6.1. *For any algorithm that assigns unique names to n processes using $s = O(n)$ TAS objects, where the initial namespace has size $M \geq n^2$ and the output namespace has size $m = O(n)$, there exists an oblivious adversary strategy that, with constant probability, forces at least one process to take $\Omega(\log \log n)$ steps.*

Proof Strategy. The proof starts with a sequence of reductions. We first reduce the problem of renaming using TAS to the problem of arranging for each process to win some TAS in a related model. We then show that every process wins a TAS only if it wins a TAS in a layered execution where each round of operations applies to locations in a new array of test-and-sets that replicates the collection used by the original algorithm.

Next, we construct an initial, independent Poisson distribution on the number of processes applying each sequence of probes. Even though processes learn information from losing test-and-sets in early rounds, by carefully pruning out processes we can restore independence between the pruned survivors. This reduces the problem to one where each class of processes loses with a fixed, independent probability in each round regardless of the actions of other processes. This is sufficient to show that some processes remain after $\Theta(\log \log n)$ rounds.

Preliminaries. We assume that processes are deterministic, and that the behavior of a process is fully determined by its initial name. We assume an oblivious adversary, so the lower bound extends to randomized algorithms by Yao's Principle [34].

The behavior of a process with a given initial name is a **type**, which specifies what operations it carries out. A type is a function from sequences of TAS return values (0 or 1) to operations TAS($T[j]$) or **return**(j), where TAS($T[j]$) applies a TAS operation to $T[j]$, $1 \leq j \leq s$, and **return**(j) returns the name j, $1 \leq j \leq m$. An algorithm is a (possibly random) assignment of types to processes. We will show a lower bound for any fixed assignment of types, and use Yao's Principle [34] to extend this to randomized algorithms.

An adversary controls the interleaving of operations in the system. The adversary is oblivious, which means that it chooses a schedule consisting of a sequence of process ids without regard to the types of the processes. At each step, the next process in the sequence carries out the operation selected by its type based on the outcome of previous operations; if this operation is **return**(j), the process chooses name j and executes only no-ops if scheduled again.

Recall that a random variable X is **Poisson** with **rate** λ if $\Pr[X = k] = e^{-\lambda}\lambda^k/k!$; we indicate this by $X \sim \text{Pois}(\lambda)$. The rate λ also gives both the expectation and variance of X. Let $P_\lambda(n) = \sum_{k=0}^{n} e^{-\lambda}\lambda^k/k!$ be the cumulative distribution function $\Pr[X \leq n]$ for $X \sim \text{Pois}(\lambda)$.

Due to space limitations, the proofs of the technical claims have been deferred to the full version of this paper.

6.1 Reductions and Adversarial Execution

To simplify the lower bound argument, we constrain the interaction between processes through a sequence of reductions. The first eliminates the distinction between TAS($T[j]$) and **return**(s) operations; in the revised problem, a process acquires a name by *winning* a TAS object. (Recall that a process **wins** a TAS object if it is the first to access it.) In the reduced problem, **return**(j) operations are replaced by TAS($T[j]$) operations on a larger array in the code of each process (which is given by its *type*).

LEMMA 6.2. *For any renaming algorithm A on s TAS objects, with an output namespace of size m, there exists an algorithm A' on $s + m$ TAS objects, such that for any schedule σ involving n processes, if every process in σ chooses a unique name when running algorithm A, every process in σ wins some TAS object in algorithm A'.*

For the second reduction, we replace the single array T of TAS objects with a sequence of arrays T_ℓ, where each array T_ℓ is of the same length as T, and the ℓ-th TAS operation by a process is always applied to an object in T_ℓ. We also show that we can assume that any process leaves the protocol immediately if it wins a TAS.

LEMMA 6.3. *Let A be an algorithm in which processes carry out operations on an array of TAS objects $T[1] \ldots T[s]$. Let A' be a modified algorithm in which a process (a) leaves the protocol immediately as soon as it wins a TAS; and (b) carries out its ℓ-th TAS operation (if it has not already left) on $T_\ell[j]$, where j is the index yielded by its type in A assuming it loses its first $\ell - 1$ TAS operations. Then if A and A' are run with the same schedule σ, the set S' of processes that appear in σ but fail to win a TAS in A' is a subset of the corresponding set S of processes that appear in σ but do not win a TAS in A.*

The Execution. We now construct a layered schedule σ such that any algorithm takes $\Omega(\log \log n)$ layers with constant probability to reduce the number of remaining processes to a constant. Each layer of σ consists of a single step by each process instance. These steps are ordered by a random permutation that is chosen uniformly and independently for each layer. Since σ does not depend on the actions of the algorithm, it can be supplied by an oblivious adversary.

The main challenge is that, in such an execution, the number of processes that have not yet won a TAS, and thus must continue, may drop very fast as we proceed through layers. To lower bound the number of processes that continue, we use a Poisson approximation (see, e.g., [30, §5.4]) to make the initial number of processes accessing each TAS independent, and apply a coupling gadget to keep the surviving processes in each layer independent of each other. We cannot apply this directly to the original process, since, for example, if p and q both access the same TAS object $T_\ell[j]$, the fact that

p lost it may increase the conditional probability that q lost it as well. We *mark* a subset of survivors for each TAS such that the counts of marked processes are independent. Marking *does* require observing the execution of the algorithm, but it is only used in the analysis and does not affect the behavior of the algorithm or the adversary.

Precisely, for $i = 1 \dots M$, let X_i^0 be independent Poisson random variables, where X_i^0 has rate λ_i^0. We interpret X_i^0 as the number of instances of process p_i that are included in the execution σ. If $X_i^0 > 1$, we are in trouble, but we ensure that the chance that this occurs is small by choosing small enough λ_i^0; the cost of infrequently generating a bad schedule will be compensated for by the useful properties of Poisson random variables that we exploit later in the proof.

We define *marked* processes to be the processes that do not win a TAS up to some point in the execution. Formally, for a layer ℓ and a process p_i, the variable X_i^ℓ indicates the number of *marked* instances of p_i. Initially, after 0 layers, all instances of all processes are marked. After $\ell \geq 1$ layers, which processes are marked is precisely determined by a procedure described in Section 6.2. Our goal is to obtain a lower bound on the number of such processes in each layer.

We will show that our marking procedure ensures that the X_i^ℓ are independent Poisson random variables, with rates that evolve predictably as a consequence of the probabilities that the various types assign to each TAS object $T_\ell[j]$. Then the total number of marked processes in each layer ℓ will also follow a Poisson distribution, with rate $\lambda^\ell = \sum_i \lambda_i^\ell$.

6.2 Execution Analysis

We start by carefully constructing a Poisson random variable Y with the property that, if we mark the last Y processes to access a TAS, then we get independent Poisson counts on the number of marked processes of each type.

LEMMA 6.4. *Fix an index set S, and let $X_i \sim \text{Pois}(\lambda_i)$ be independent random variables, for all $i \in S$. Let $Z = \sum_{i \in S} X_i \sim \text{Pois}(\lambda)$, where $\lambda = \sum_{i \in S} \lambda_i$. Let $Y \sim \text{Pois}(\gamma)$ be a random variable that is coupled with Z such that $Y \leq \max(0, Z-1)$ always and Y is conditionally independent of the X_i conditioned on Z. Choose a permutation π of a string σ consisting of X_i instances of each i in S uniformly at random. Let X_i' for each i in S be the number of instances of i that occur in the last Y positions in π. Then the X_i' are independent Poisson random variables with $X_i' \leq X_1$, and $X_i' \sim \text{Pois}\left(\lambda_i \cdot \frac{\gamma}{\lambda}\right)$.*

We will now show that for every $Z \sim \text{Pois}(\lambda)$, there is a coupled random variable $Y \sim \text{Pois}(\min(\lambda^2/4, \lambda/4))$ with $Y \leq \max(0, X-1)$ always. Since our construction of Y does not depend on the decomposition of Z into X_i, it can also be made to have the conditional independence property required by Lemma 6.4. To demonstrate the existence of the desired Y, we consider the cumulative distribution functions of Z and Y. We prove the following.

LEMMA 6.5. *For all $n \in \mathbb{N}$ and all $\lambda \geq 0$, $P_\lambda(n+1) \leq P_{\min(\lambda^2/4, \lambda/4)}(n)$.*

Marked processes. We now have the machinery we need to characterize how many processes are marked at each layer. We first describe the marking procedure precisely.

In layer ℓ, for each TAS object $T_\ell[j]$ there is a set of types S_j^ℓ that apply an operation to $T_\ell[j]$. Let $Z_j^\ell = \sum_{i \in S_j^\ell} X_i^\ell$ be the number of marked processes that access $T_\ell[j]$, and let $Y_j^\ell \leq \max(0, X-1)$ be the coupled Poisson variable whose existence is implied by Lemma 6.5. Let the last Y_j^ℓ marked processes to access $T_\ell[j]$ keep their marks for the next round; note that because $Y_j^\ell < Z_j^\ell$ when Z_j^ℓ is nonzero, none of these processes can be the first to access T_j^ℓ. From Lemma 6.4, the counts $X_i^{\ell+1}$ of processes of each type that retain their marks are independent Poisson random variables, and the rate $\lambda_i^{\ell+1}$ of X_i^ℓ is equal to $\lambda_i^\ell \left(\mathbf{E}[Y_j^\ell] / \mathbf{E}[Z_j^\ell]\right)$ where type i accesses $T_\ell[j]$ in layer ℓ.

We now show that no matter how types choose TAS objects, the total expected number of marked processes does not drop too fast from layer ℓ to layer $\ell+1$.

LEMMA 6.6. *Let s be the number of TAS objects in layer ℓ. If $\lambda^\ell \leq s/2$ then $\lambda^{\ell+1} \geq \frac{(\lambda^\ell)^2}{4s}$. If $\lambda^\ell > s/2$ then $\lambda^{\ell+1} \geq \frac{\lambda^\ell}{4}$.*

Final Argument. We now complete the proof of Theorem 6.1. Assume an algorithm A that assigns unique names from 1 to $m = O(n)$ to n processes out of an initial namespace of $M \geq n^2$ using $s = O(n)$ TAS objects. Note that m and s must both be at least n.

To build the adversarial execution, we choose $X_i^0 \sim \text{Pois}(n/2M)$ initial instances of each process i, so that $\lambda^0 = n/2$, and construct the rest of the layered execution σ with $s+m$ TAS objects per layer as described in the preceding sections.

Let $r^\ell = \lambda^\ell/(s+m)$ be the ratio between the total expected number of marked processes after ℓ layers and the number of TAS objects in each layer. For $\ell = 0$ this gives $r^0 = (n/2)/(s+m)$, which is both $\Omega(1)$ and bounded above by $1/4$. For $\ell \geq 1$, $r^\ell \leq r^0 \leq 1/4$, which implies $\lambda^\ell = (s+m)r^\ell \leq (s+m)/2$. Lemma 6.6 then shows that $r^{\ell+1} = \lambda^{\ell+1}/(s+m) \geq (\lambda^\ell)^2/4(s+m)^2 = ((s+m)r^\ell)^2/4(s+m)^2 = (r^\ell)^2/4$.

We solve the recurrence to get $r^\ell \geq (r^0)^{2^\ell}/4^{(2^\ell-1)} = 4(r^0/4)^{2^\ell}$. Choosing $\ell = \lfloor \lg\lg(s+m) + \lg\lg(4/r_0) \rfloor = \Omega(\log\log n)$ gives $r^\ell \geq 4(r^0/4)^{\lg(s+m)\lg(4/r_0)} = 4/(s+m)$. Hence the expected number of surviving processes at layer $\ell = \Omega(\log\log n)$ is $\lambda^\ell \geq 4$.

To complete the argument, we apply the union bound to all the ways in which the procedure generating the schedule may fail. First, our initial choice of processes might include more than n processes. Since $\mathbf{E}[X^0] = \lambda^0 = n/2$, this occurs with probability at most $1/2$. Second, our initial choice of processes might include two or more copies of the same process. For each type i, $\Pr[X_i^0 \geq 2] = 1 - e^{-\lambda_i^0}(1+\lambda_i^0) \leq 1 - (1-\lambda_i^0)(1+\lambda_i^0) = (\lambda_i^0)^2 = (n/2M)^2$. Summing the bound over all M types gives a bound of $M(n/2M)^2 = n^2/4M \leq 1/4$ on the probability that any of these events occur. Finally, we must consider the possibility that there are no marked processes after round ℓ. This occurs with probability $e^{-\lambda^\ell} \leq e^{-4}$.

It follows that we get an execution with at least one marked process (and thus at least one process that has not yet acquired a name) after $\Omega(\log\log n)$ layers with probability at least $1 - 1/2 - 1/4 - e^{-4} \geq 0.23168 = \Omega(1)$. This concludes the proof of Theorem 6.1.

7. CONCLUSION AND FUTURE WORK

We presented sub-logarithmic randomized algorithms for loose renaming against a strong adversarial scheduler, and

a lower bound suggesting that $\Omega(\log \log n)$ is an inherent threshold when using linear space. Our algorithms circumvent the classic logarithmic information-based lower bounds, e.g., [29], either by exploiting extra information about maximal contention n, or by allowing for error in the namespace size. Thus, a natural extension of our work would be to exploit these ideas for sub-logarithmic implementations of other concurrent data structures. Additional directions for future work would be to improve the individual step complexity for adaptive renaming, and to generalize the lower bound technique.

8. REFERENCES

[1] Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-lived renaming made adaptive. In *Proc. of 18th PODC*, pages 91–103, 1999.

[2] Y. Afek and M. Merritt. Fast, wait-free $(2k-1)$-renaming. In *Proc. of 18th PODC*, pages 105–112, 1999.

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

[4] D. Alistarh, J. Aspnes, K. Censor-Hillel, S. Gilbert, and M. Zadimoghaddam. Optimal-time adaptive strong renaming, with applications to counting. In *Proc. of 30th PODC*, pages 239–248, 2011.

[5] D. Alistarh, J. Aspnes, S. Gilbert, and R. Guerraoui. The complexity of renaming. In *Proc. of 52nd FOCS*, pages 718–727, 2011.

[6] 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.

[7] J. Aspnes. Faster randomized consensus with an oblivious adversary. In *Proc. of 31st PODC*, pages 1–8, 2012.

[8] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *J. of the ACM*, 37(3):524–548, 1990.

[9] H. Attiya and T. Djerassi-Shintel. Time bounds for decision problems in the presence of timing uncertainty and failures. *J. of Parallel Distrib. Comp.*, 61(8):1096–1109, 2001.

[10] H. Attiya and A. Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. on Comp.*, 31(2):642–664, 2001.

[11] H. Attiya and A. Paz. Counting-based impossibility proofs for renaming and set agreement. In *Proc. of 26th DISC*, pages 356–370, 2012.

[12] Y. Azar, A. Broder, A. Karlin, and E. Upfal. Balanced allocations. *SIAM J. on Comp.*, 29(1):180–200, 1999.

[13] A. Bar-Noy and D. Dolev. Shared-memory vs. message-passing in an asynchronous distributed environment. In *Proc. of 8th PODC*, pages 307–318, 1989.

[14] M. Bender and S. Gilbert. Mutual exclusion with $O(\log^2 \log n)$ amortized work. In *Proc. of 52nd FOCS*, pages 728–737, 2011.

[15] A. Broder and A. Karlin. Multilevel adaptive hashing. In *Proc. of 1st SODA*, pages 43–53, 1990.

[16] A. Brodsky, F. Ellen, and P. Woelfel. Fully-adaptive algorithms for long-lived renaming. *Distr. Comp.*, 24(2):119–134, 2011.

[17] J. Burns and G. Peterson. The ambiguity of choosing. In *Proc. of 8th PODC*, pages 145–157, 1989.

[18] A. Castañeda and S. Rajsbaum. New combinatorial topology bounds for renaming: the lower bound. *Distr. Comp.*, 22(5-6):287–301, 2010.

[19] A. Castañeda and S. Rajsbaum. New combinatorial topology bounds for renaming: The upper bound. *J. of the ACM*, 59(1):3, 2012.

[20] 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.

[21] D. Fotakis, R. Pagh, P. Sanders, and P. G. Spirakis. Space efficient hash tables with worst case constant access time. *Theory of Comp. Syst.*, 38(2):229–248, 2005.

[22] G. Giakkoupis and P. Woelfel. On the time and space complexity of randomized test-and-set. In *Proc. of 31st PODC*, pages 19–28, 2012.

[23] G. Giakkoupis and P. Woelfel. A tight RMR lower bound for randomized mutual exclusion. In *Proc. of 44th ACM STOC*, pages 983–1002, 2012.

[24] W. Golab, L. Higham, and P. Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *Proc. of 43rd ACM STOC*, pages 373–382, 2011.

[25] D. Hendler and P. Woelfel. Adaptive randomized mutual exclusion in sub-logarithmic expected time. In *Proc. of 29th PODC*, pages 141–150, 2010.

[26] D. Hendler and P. Woelfel. Randomized mutual exclusion with sub-logarithmic RMR-complexity. *Distr. Comp.*, 24(1):3–19, 2011.

[27] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proc. of 16th DISC*, pages 339–353, 2002.

[28] M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *J. of the ACM*, 46(2):858–923, 1999.

[29] P. Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In *Proc. of 17th PODC*, pages 201–210, 1998.

[30] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[31] M. Moir and J. H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Program.*, 25(1):1–39, 1995.

[32] A. Panconesi, M. Papatriantafilou, P. Tsigas, and P. M. B. Vitányi. Randomized naming using wait-free shared variables. *Distr. Comp.*, 11(3):113–124, 1998.

[33] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. of the ACM*, 27(2):228–234, 1980.

[34] A. C.-C. Yao. Probabilistic computations: Toward a unified measure of complexity. In *Proc. of 18th FOCS*, pages 222–227, 1977.