Fast Randomized Test-and-Set and Renaming

Dan Alistarh[1], Hagit Attiya[1,2], Seth Gilbert[1], Andrei Giurgiu[1], and Rachid Guerraoui[1]

[1] Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
[2] Technion, Haifa, Israel

Abstract. Most people believe that renaming is easy: simply choose a name *at random*; if more than one process selects the same name, then try again. We highlight the issues that occur when trying to implement such a scheme and shed new light on the read-write complexity of randomized renaming in an asynchronous environment. At the heart of our new perspective stands an adaptive implementation of a randomized test-and-set object, that has poly-logarithmic step complexity per operation, with high probability. Interestingly, our implementation is anonymous, as it does not require process identifiers. Based on this implementation, we present two new randomized renaming algorithms. The first ensures a tight namespace of n names using $O(n \log^4 n)$ total steps, with high probability. This improves on the best previously known algorithm by almost a quadratic factor. The second algorithm achieves a namespace of size $k(1+\epsilon)$ using $O(k \log^4 k / \log^2(1+\epsilon))$ total steps, both with high probability, where k is the total contention in the execution. It is the first *adaptive* randomized renaming algorithm, and it improves on existing deterministic solutions by providing a smaller namespace, and by significantly lowering complexity.

1 Introduction

Names, or identifiers, are instrumental for efficiently solving a variety of problems that arise in distributed systems. And yet, in many cases, names are not available. Participants may be anonymous, or may wish to hide their true identity for reasons of privacy. Alternatively, participants may have names, but they may be taken from a very large namespace. For example, nearly every networked device has an ethernet address, and yet the namespace is so large as to reduce the usefulness of such names. Thus, a significant amount of research (e.g., [4, 13, 20, 24, 25]) has analyzed the feasibility and complexity of the *renaming* problem in a crash-prone distributed system.

Unfortunately, renaming in a fault-prone system can be expensive, if not impossible. For example, wait-free *tight* renaming—where the namespace exactly matches the set of n participants—is impossible for deterministic algorithms that tolerate crash failures [6, 13, 20]. Even *loose* renaming, where the namespace is of size $(2n-1)$, can be quite expensive, as the best known solutions require at least $\Theta(n^3)$ total steps [2, 24].

Yet in practice, most people believe that renaming is relatively easy: simply choose a name *at random*; if more than one process selects the same name, then try again. Several subtle problems occur when trying to implement such a scheme:

– From how big a namespace should the name be chosen? If the namespace is large, say $\Theta(n^2)$, where n is the number of participants, then such schemes are trivial.

However, when the namespace is smaller—e.g., $(1+\epsilon)n$, for some $0 < \epsilon < 1$—the efficiency is less clear. And when the namespace is tight, i.e., of size precisely n, then some participants may have to retry repeatedly in order to find a free name.

- How does a participant claim a name? How does a participant determine whether its chosen name is unique? Effectively, when more than one participant selects the same name, participants must agree on which participant wins the name. Such agreement must be fault-tolerant, i.e., succeed even if processes fail; and it must be irrevocable, meaning that once a participant is assigned a name, it cannot later be forced to abandon it. The simple solution would be to run a distributed *consensus* algorithm to agree on which process owns each name. However, asynchronous, wait-free deterministic consensus is impossible [16], and the randomized version is inherently expensive, requiring $\Omega(n^2)$ total steps [7].
- How are processes scheduled? If the processes are scheduled in a synchronous fashion, then resolving contention among processes may not be difficult. However, in an asynchronous system, processes can be scheduled in any order. Worse, for a strong (adaptive) adversarial scheduler, the choice of a schedule may depend on the random choices being made by the processes. Thus the random choices made by the processes are not entirely independent in the usual sense.

In this paper, we present two efficient *randomized* renaming algorithms for an asynchronous, fault-prone system subject to a strong, adaptive adversary.

The key building block for both algorithms is a new efficient implementation of a randomized test-and-set object. This algorithm answers the question of how a process can claim a name: a test-and-set object allows multiple processes to compete (for example, for a name), ensuring that there will be exactly one winner. The algorithm, which we call RatRace, is more efficient than consensus [7]: if there are k competitors, the total step complexity for a test-and-set is $O(k \log^2 k)$ read/write operations, with high probability. Of note, the RatRace algorithm is *adaptive*: the step complexity depends on k, the actual number of competitors (not on n, the total number of possible competitors). The algorithm efficiently combines the idea of a randomized splitter tree, first used in [9], with the tournament tree algorithm by Afek et al. [1]. Our renaming algorithms rely on both the *adaptivity* and *anonymity* properties of this implementation. Given the power of test-and-set to simplify coordination in a distributed system, and the efficiency of our solution, we expect that the RatRace algorithm may well be useful in other settings as well.

Tight Renaming. Our first renaming algorithm, called ReShuffle, produces unique names from a *tight* namespace of n names, using $O(n \log^4 n)$ total steps (reads and writes), with high probability. The algorithm uses a simple random process to compete for names: each process repeatedly chooses a name at random, and attempts to claim it via a randomized test-and-set, stopping when it wins a name.

While the scheme is surprisingly simple, its analysis is rendered non-trivial by the fact that the scheduling and the failure pattern are controlled by the strong, adaptive adversary. For example, the adversary may look at a process's random choices prior to deciding whether it should be scheduled to perform a read or write operation. So the adversary might attempt to delay each process that chooses an unclaimed name

until there are several other processes competing for the same name; since only one process can win, the adversary can, in this way, create a significant number of wasted steps. Since the schedule depends on the random choices, we cannot treat the processes' random steps as being uncorrelated, as needed for a standard analysis. We overcome this difficulty by carefully assessing the number of extra steps that the algorithm has to take because of adversarial scheduling or crashes.

Our algorithm improves significantly on the total step complexity of previous randomized or deterministic namespace-optimal implementations [2, 15], which have at least $\Theta(n^3)$ total step complexity. It guarantees unique names in a range from 1 to n in every execution, and terminates with probability 1.

Adaptive Renaming. Our second renaming algorithm, called AdaptiveSearch, is the first randomized *adaptive* renaming solution. That is, the algorithm's namespace and complexity depend on k, the number of processes competing, rather than n, the total number of possible participants. Given any constant $\epsilon > 0$, the AdaptiveSearch algorithm ensures unique names from a namespace of size $k(1+\epsilon)$, where k is the total contention in the current execution, using $O(k \log^4 k / \log^2(1+\epsilon))$ total steps, both with high probability. The main idea behind the algorithm is that processes try to acquire a name in intervals of increasing size; we prove that contention in intervals towards the edge of the namespace is low enough so that each process is successful with high probability. The algorithm improves on the adaptive deterministic solutions known so far by providing a smaller namespace than is otherwise feasible, and by improving the total step complexity. (The most efficient adaptive algorithm to date [14] has total step complexity $O(k^2)$, and renames in $(8k - \log k - 1)$ names.)

Discussion. Both algorithms are within logarithmic factors from the immediate lower bound of $\Omega(n)$ (or $\Omega(k)$) on the total step complexity of renaming. They also have the interesting property that they do not require unique process identifiers at the beginning of the execution—as such, the algorithms are *anonymous*.

Also of note, the ReShuffle algorithm is the first randomized algorithm to attain tight renaming with total step complexity less than $\Theta(n^2)$. Since $\Omega(n^2)$ is known to be the (tight) lower bound for the step complexity of randomized *consensus* [7], our algorithm yields the first clear separation in terms of complexity between randomized tight renaming and randomized consensus in asynchronous shared-memory.

The impossibility of wait-free renaming in a namespace smaller than $(2n-1)$ [13, 20] is circumvented by the use of randomization. There exist infinite length executions of infinitesimal probability weight, in which the algorithms do not terminate. Also, note that our test-and-set implementation cannot solve consensus for more than two processes, which is why it is not affected by the $\Omega(n^2)$ lower bound shown in [7].

Roadmap. In Section 2, we present the model and the problem statement, while Section 3 presents a detailed account of related work. Section 4 presents the implementation of adaptive test-and-set. Based on this, we introduce the ReShuffle algorithm in Section 5, and analyze its complexity. Section 6 presents the adaptive renaming implementation. We conclude in Section 7, stating some limitations of our approach, together

with a host of open problems. Due to space constraints, some of the proofs have been deferred to the full version of this paper.

2 Model and Problem Statement

We assume an asynchronous shared memory model with n processes, $t < n$ of which may fail by crashing. Let M be the size of the space of initial identifiers that processes in the system may have[3]. For the adaptive algorithm, we consider k to denote total *contention*, i.e. the total number of processes that take steps during a certain execution. We assume that processes know n, but do not know k. Processes communicate through multiple-writer-multiple-reader atomic registers. Our algorithms are randomized, in that the processes' actions may depend on random local coin flips. We assume that the process failures and the scheduling are controlled by a strong adaptive adversary. In particular, the adversary knows the results of the random coin flips that the processes make and can adjust the schedule and the failure pattern accordingly.

In this context, the *renaming problem* requires that each correct process should eventually return a name, and the names returned should be unique. The size of the resulting namespace should only depend on n and on t. Note that, in our algorithms, we relax the assumption of unique initial identifiers, made in the original problem statement [6]. We assume $t \leq n - 1$, hence our solutions are *wait-free*. The complexity of our solutions is measured in terms of total steps (reads and writes, including random coin flips).

In the following, we say that an event happens "with high probability" (whp) if it occurs with probability $\geq 1 - 1/n^c$, with $c \geq 1$ constant. In the case of the adaptive algorithms, the probability bound is at least $\geq 1-1/k^c$, with $c \geq 1$. Note that the failure probability in the adaptive case may be tuned to depend on n, at the cost of increased complexity (i.e., a $\log n$ factor).

3 Related Work

Our test-and-set implementation re-uses ideas from the efficient randomized collect algorithm of Attiya et al. [9] and from the wait-free implementation of randomized test-and-set by Afek et al. [1]. We make use of the splitter object, originally introduced in [24], and its randomized version introduced in [9]. Overall, the structure of RatRace is similar to the adaptive algorithm for mutual exclusion by Anderson et al. [3], although the problem and the fault model we analyze are different. We use the two-process randomized test-and-set algorithm by Tromp and Vitànyi [28] as a building block.

The renaming problem has been introduced by Attiya et al. [6]. In the original paper, the authors present a wait-free solution using $(2n - 1)$ names in an asynchronous

[3] Note that some earlier work (e.g., [15]), uses n to denote the total number of identifiers that processes may have, which may also be seen as the maximum total number of processes in the system. They use k for the maximum number of processes that may participate in an execution (which we denote by n).

message-passing system, and show that at least $(n+1)$ names are required in the wait-free case. The lower bound was improved to $(2n-2)$ in a landmark paper by Herlihy and Shavit [20]. Recent work by Rajsbaum and Castañeda [13] shows that deterministic wait-free renaming may be possible for $\leq (2n-2)$ names for specific parameter values.

The complexity of deterministic shared-memory renaming implementations has been an active research topic. Burns and Peterson [12], Borowski and Gafni [10], Anderson and Moir [24], Moir and Garay [25] were among the first to propose wait-free deterministic renaming algorithms into a namespace of size $(2n-1)$. These solutions have very high total step complexity; for some, the total step complexity is exponential (e.g. [12, 25]). Anderson and Moir [4] propose a variant of renaming that attains a tight namespace of n names using stronger *set-first-zero* objects. Note that their algorithm could be rephrased using our one-shot test-and-set implementation, although it would have at least total $\Theta(n^3)$ total step complexity.

Later work analyzed *adaptive* renaming algorithms, in which the step complexity and the size of the namespace depend only on total contention k, not on the maximum number of participating processes n. The first adaptive algorithm was introduced by Attiya and Fouren [8]. They achieve a namespace of $(6k-1)$ names, with a total complexity of $O(k^2 \log k)$. Afek and Merritt [2] build on the previous algorithm in order to achieve adaptive wait-free $(2k-1)$-renaming with total step complexity $O(k^3)$.

In a recent paper, Chlebus and Kowalski [14] improve the complexity bounds for deterministic renaming by providing a non-adaptive implementation with local step complexity roughly $O(\log n \log M)$, renaming into a namespace of size $O(n)$. The local step complexity of their algorithm is better than that of ReShuffle, although we achieve a tight namespace of n names, and comparable total step complexity. They also introduce an adaptive implementation with $O(k)$ local step complexity, which achieves renaming in $8k - \log k - 1$ names, and show the first non-trivial deterministic lower bound on step complexity, of $(1 + \min(k-2, \log_{2r} \frac{M}{2T}))$, where r is the number of shared registers used by the algorithm, and T is the size of the target namespace to rename into. One of the advantages of the algorithms from this reference is that they use little total memory $O(n \log(M/n))$. In comparison, our algorithms pre-allocate $O(n^2)$ memory, and use $O(n \text{ polylog } n)$ total memory, without assuming any bound M on the initial namespace.

Recent work by Ellen et al. [11] analyzes the complexity of *long-lived* adaptive renaming (i.e., processes may release their names) in shared-memory, under various synchrony assumptions. Their asynchronous algorithm ensures $\Theta(k)$ overhead for acquiring a new name, although assumes that stronger LL/SC primitives are available; hence their results are not directly comparable with ours. This reference also contains an excellent overview of prior work on renaming.

The feasibility of *randomized* renaming in an asynchronous system has been first considered by Panconesi et al. [27]. They present a wait-free solution that ensures a namespace of size $n(1+\epsilon)$ for $\epsilon > 0$, with expected $O(M \log^2 n)$ total step complexity, using only single-writer multiple-reader registers. Their strategy is similar to that of this paper: they introduce a one-shot test-and-set implementation, and processes obtain names based on which test-and-set they manage to acquire. Note that their test-and-set implementation is not adaptive, which is why the complexity of the solution depends

on M. Moreover, the namespace they obtain is not tight. Interestingly, a strategy similar to that of ReShuffle is mentioned in this reference (Section 4.1), but is considered "too hard to analyze." Note that our adaptive algorithm uses a different strategy than that of this reference, although the bounds on the namespace size look similar.

The second paper to analyze randomized renaming is by Eberly et al. [15]. The authors obtain a *tight* non-adaptive renaming algorithm based on the randomized wait-free implementation of test-and-set by Afek et al. [1]. Their algorithm is long-lived, and is shown to have amortized step complexity of $O(n \log n)$ per process, under a given cost measure. However, a simple analysis shows that their algorithm has average-case total step complexity of at least $\Theta(n^3)$, even if processes do not release their names.

4 An Adaptive Test-and-Set Implementation

We start by presenting an adaptive one-shot implementation of a randomized adaptive test-and-set object. The object exports a single Test-and-Set operation, whose sequential specification is provided in Figure 2.

Note that one-shot test-and-set cannot be implemented deterministically wait-free in asynchronous shared memory, since it has consensus number 2 (see [19] for details). We present an efficient randomized implementation that guarantees the desired properties with probability 1, and is linearizable, following the definition in [21]. Our implementation is *adaptive*, in that the complexity of an operation depends on the contention k at the object, and not on n, the total number of processes.

4.1 The RatRace Algorithm

The RatRace implements the one-shot test-and-set object as defined above. Any operation on the object has step complexity $O(\log^2 k)$ per process with high probability, where k denotes the total contention at the object. The algorithm pre-allocates $O(n^3)$ memory, and uses $O(k)$ memory with high probability. A sketch of the algorithm's structure can be found in Figure 1.

Algorithm Structure. We begin from a binary tree of randomized splitters (as previously defined in [9]), of height $3 \log n$, which we call the *primary tree*. Each process starts the algorithm at the root splitter in the primary tree; if it does not manage to acquire the current splitter, it goes either left or right, each with probability $1/2$, until it manages to acquire a splitter. If a process reaches a leaf of the primary tree without having acquired a splitter, it accesses a *backup grid*, which we describe in the next paragraph. To simplify the exposition, assume that, in this execution, all processes either obtained randomized splitters in the first tree, or crashed.

Once it managed to obtain a splitter, the process tries to work its way up back to the root, through a series of three-process "tournaments," one at each splitter node. Each splitter in the primary tree has associated with it a three-player "tournament," which is played between the owner of the splitter and the winners of the three-player test-and-sets corresponding to the two child nodes of the splitter. A three-player test-and-set is decided as follows: the two child nodes play each other, and the owner of the

current splitter plays the winner of the first match. Each two-player match is decided using the randomized two-process test-and-set algorithm of Tromp and Vitànyi [28]. (Alternatively, we could use a randomized consensus algorithm with $n = 2$, e.g. [5], although the properties stay the same.) Note that the matches are decided in a wait-free manner, since a process wins automatically if the opponent does not show up.

The Backup Grid. The backup grid is an $n \times n$ grid of *deterministic* splitters, identical to that of Anderson and Moir [4], where the two children of a splitter are the splitter to its right, and the one below. Each process starts the backup algorithm at the top left splitter. As such, the structure guarantees that any correct process that accesses it eventually acquires a deterministic splitter. Just as in the previous case, once a process acquires a splitter, it tries to backtrack to the entry point through a series of three-player test-and-sets. The winner of the test-and-set at the entry splitter is also the winner of the backup grid.

Decision. The winner of the three-player test-and-set at the root of the primary tree plays the winner of the entry splitter in the backup grid. The winner of this last match returns winner. Every process that loses in a three-player test-and-set returns loser.

Linearization. In order to maintain the linearization guarantees of the test-and-set object, a process that loses a three-player test-and-set writes true to a multi-writer-multi-reader $Resolved$ register associated with the root of the primary tree, *before* returning loser. Processes read the register as the first step in their Test-and-Set invocation: if they read true, they automatically return loser.

4.2 Analysis of the RatRace Algorithm

It is relatively straightforward to check that the RatRace algorithm guarantees the correctness properties of the *test-and-set* object as stated in Section 4, therefore we omit the proof from this extended abstract. Termination with probability 1 is ensured since we use wait-free elements and the two-process test-and-set algorithm of [28], which terminates with probability 1. We next focus on the linearizability of the implementation, and on its performance in terms of total step complexity. Our first result shows that our implementation is linearizable, in the sense of Herlihy and Wing [21]. The proof is based on the observation that, before a loser indication is returned by RatRace, a potential winner has to take at least one step in the algorithm.

Lemma 1 (Linearization). *The* RatRace *algorithm is* linearizable*: for every execution of* RatRace*, there exists a total order over all the complete* Test-and-Set *operations together with a subset of the incomplete* Test-and-Set *operations such that every operation is immediately (atomically) followed by a response, and the sequence of operations given by that total order is consistent with a sequential execution of a test-and-set object, i.e. the order respects the real-time order of non-overlapping operations.*

We now analyze the performance of RatRace. Let k denote the number of processes that enter the RatRace in an execution $\mathcal{E}$, i.e. the total contention. The next result states

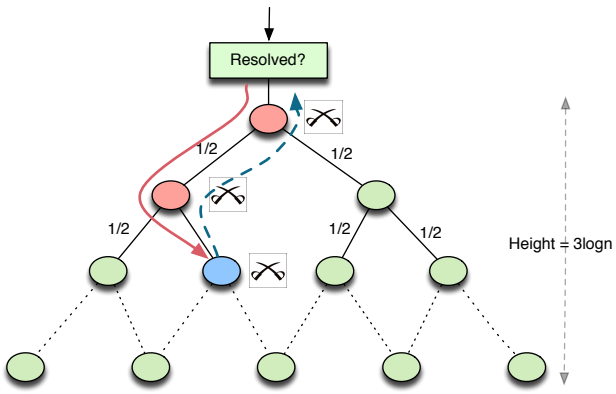

Fig. 1. Structure of the RatRace protocol. A process first checks the $Resolved$ register, and then walks down the randomized splitter tree trying to acquire a splitter. In this figure, the process followed the solid path and acquired a splitter. Next, the process works its way back up the tree, participating in three-player tournaments at each node (the dotted path). If it loses along this path, then it marks the $Resolved$ register and returns loser. Otherwise, if it wins the root tournament, it plays the winner from the backup grid (not shown here). The winner of this last match returns winner.

that, with high probability, every process acquires a splitter in the primary tree. As a consequence of this fact, for the rest of the performance analysis, we will assume that all processes acquire nodes in the primary tree, since the backup case is extremely unlikely. We provide the intuition for why this holds; an exact proof follows from the analysis in [9], Lemma 8.

Lemma 2. *The probability that there exists a process p that does not acquire a randomized splitter in the primary tree of the* RatRace *object is at most $1/n$.*

Proof (Sketch). Let q be a process that does not manage to acquire any splitter in the primary tree. Hence, q did not manage to acquire the leaf splitter it reached. Since a process always acquires a splitter if it accesses it alone, this implies that another process q' accessed the same leaf splitter. However, the leaf splitter is accessed by q as a consequence of $3 \log n$ random choices of bits. Hence process q' must have performed the exact same random choices. Since the choices are independent, the probability that this occurs is $(1/2)^{3 \log n} = 1/n^3$. Hence the probability that there exists a process that performs exactly the same random choices as q is at most $1/n^2$. By the union bound, it follows that the probability that there exists a process p that "falls off" the primary tree is at most $1/n$. □

Let the *active primary tree* denote the minimum subtree of the primary tree containing all splitters that are acquired in the execution. The second result bounds the number of nodes in the active primary tree, and shows that the tree is well balanced, with high probability. The proof is similar to that of Lemma 2. For a complete argument, please

see reference [9], Lemma 11. Note that this lemma also bounds the space complexity the primary tree.

Lemma 3. *The number of nodes in the* active primary tree *is at most* $7k$*, and its height is at most* $3 \log k$*, both with high probability.*

Next, we look at the read-write complexity of the two-process test-and-set algorithm of Tromp and Vitànyi [28] that we use to decide the two-process games. The following bounds follow from an analysis of the algorithm.

Lemma 4. *The randomized two-process test-and-set algorithm of [28] has expected constant read-write complexity, and performs less than* $\alpha \log k$ *reads and writes with high probability, for a constant* $\alpha > 1$.

Proof (Sketch). Please recall that the algorithm of [28] is composed of asynchronous "rounds" of computation, and performs a constant number, say β, of reads and writes per round. In every round, the probability of success is $1/2$. Thus, the expected step complexity is constant. The probability that the algorithm performs at least $2\beta \log k$ total steps is at most $1/k^{2\beta}$, from which the claim follows. □

The next result analyzes the total step complexity of RatRace.

Lemma 5. *The* RatRace *algorithm uses* $O(\log^2 k)$ *steps per process, with high probability. Hence, the total step complexity is* $O(k \log^2 k)$*, with high probability.*

Proof. Without loss of generality, we analyze the number of steps performed by a winning process. First note that, by Lemma 2, it is enough to bound the complexity in the case where the process only accesses the primary tree. By Lemma 3, a process performs $O(\log k)$ steps, with high probability, when going down the tree in order to acquire a randomized splitter, since each splitter has constant step complexity. When climbing back up, the process may play up to $O(\log k)$ three-player test-and set games. By Lemma 4, we obtain that the process performs up to $O(\log^2 k)$ steps, with high probability. □

5 A Randomized Algorithm for Tight Renaming

In this section, we present ReShuffle, a randomized algorithm which ensures tight renaming using $O(n \log^4 n)$ total steps, with high probability. The pseudocode of the algorithm can be found in Figure 3.

5.1 The ReShuffle Algorithm

The n processes share n test-and-set objects, each implemented using the RatRace algorithm. These shared objects are numbered from 1 to n. Computation proceeds in local phases. In each phase, the process chooses uniformly at random a test-and-set from 1 to n that it has not chosen previously, and competes in it. If the process wins the test-and-set (i.e., the chosen RatRace instance returns winner), then it takes the number associated with the test-and-set as a name and returns. Otherwise, if it lost the test-and-set, the process marks the current test-and-set as lost, and tries again in the next phase.

```
1 Variable:
2  Value, a binary MWMR atomic register,
   initially ⊥
3 procedure Test-and-Set()
4    if Value = ⊥ then
5         Value ← 1
6         return winner
7    else
8         return loser
```

Fig. 2. Sequential specification of a one-shot test-and-set object.

```
1 Shared:
2 TS[], a vector of n RatRace objects
3 procedure rename( n )
4    List ← {1, 2, ..., n}
5    while true do
6         try ← element uniformly at
          random from List
7         res ← TS[try].test-and-set()
8         if res ← winner then return try
9         else List ← List \ {try}
```

Fig. 3. The ReShuffle algorithm.

5.2 Analysis of ReShuffle

In this section, we analyze the correctness of the algorithm and its performance guarantees. First, note that name uniqueness is satisfied trivially, since a process stops after it has won its first test-and-set, and no two processes may win the same test-and-set object. We first show termination with probability 1. The proof is based on the observation that if a process accesses all n test-and-set objects, it will certainly win one of them, and hence terminate. The latter claim is based on the linearizability of our test-and-set implementation.

Lemma 6 (Termination). *With probability* 1*, each correct process eventually returns from* ReShuffle*.*

The next Theorem provides precise bounds for the total step complexity of ReShuffle. This is the main technical result of this paper. Due to space restrictions, we only provide a detailed sketch of the proof in this extended abstract.

Theorem 1 (Complexity). *The total step complexity of* ReShuffle *is* $O(n \log^4 n)$ *with high probability.*

Proof (Sketch). The first idea in the proof is to consider the total number of Test-and-Set calls (or accesses) that the processes perform as part of ReShuffle. We will consider all the accesses in their *linearization order* over all n test-and-set objects. Note that such an order exists, and is coherent at each object, because each test-and-set object is linearizable (by Lemma 1), and thus the objects are composable (or *local* [21]). We will show that the algorithm performs $O(n \log^2 n)$ total accesses in any execution, with high probability. To simplify the exposition, we modify the algorithm so that processes always pick the next test-and-set to access uniformly at random, without discarding test-and-set objects that have been accessed previously. We can see ReShuffle as a slightly more efficient version of this simplified scheme, in which a process receives immediately a loser indication if its random choice indicates a test-and-set object that it has accessed before.

Fix a constant $\alpha > 4$. We show that if the algorithm performs more than $\alpha n \log^2 n$ total Test-and-Set accesses during an execution, then, with high probability, each test-and-set object is accessed at least once.

A tempting, yet unsuccessful approach to bound the total number of calls before each test-and-set is accessed once would be to use the well-known *coupon collector* process [23, 26], which guarantees that n distinct coupons will be discovered using $O(n \log n)$ independent random trials. Note, however, that the strong adversary controls the scheduling of the trials, which causes this simple version of the analysis to fail. Our analysis takes this factor into account, and proves that, even though the adversary may re-order calls using its knowledge of the processes' random choices, all the objects are accessed after $O(n \log^2 n)$ random calls.

Let U to be the number of test-and-set objects that have not been accessed by any process, at a certain point in the execution. We split the execution into phases. For $1 \leq i \leq \log n$, we define phase i as the time interval in which $n/2^{i-1} \geq U > n/2^i$. (Recall that we consider the linearized execution.) We prove that, by performing $\alpha n \log n$ total test-and-set accesses, the algorithm progresses for at least one phase, with high probability. Also, the number of processes that take steps in phase i or later is at most $n/2^i$, with high probability.

We proceed by induction. In this sketch of proof, we only consider the induction step (the base case is similar). Assume that the claim holds at all phases $\leq i$, and we prove that it also holds at phase $i+1$. First note that, if the adversary schedules at most $\alpha n \log n$ processes to access test-and-set objects during phase $i+1$, then the processes will make at most $n/2^i + \alpha n \log n$ total random choices during this phase. This is because, by the induction step, there are at most $n/2^i$ processes that have not terminated up to phase $i+1$, and each of them might make a random choice in this phase prior to accessing a test-and-set object. Also, for every access of a test-and-set that the adversary schedules, at most one more choice is made.

We will show that, since these choices are uniformly random, it is extremely improbable that the adversary finds $\alpha n \log n$ random choices made in this phase, which it can schedule without allowing the algorithm to move to the next phase. Let D_i be the set of test-and-set objects accessed prior to the beginning of phase $i+1$. Notice that the algorithm stays in phase $i+1$ after $\alpha n \log n$ total accesses if there exist 1) a set C of $\alpha n \log n$ random choices made during this phase, and 2) a set S of less than $n/2^{i+1}$ test-and-set objects not in D_i, such that all the choices in C are made on test-and-set objects from S, or on the $n(1-2^i)$ test-and-set objects in D_i. (Note that this formulation slightly increases the power of the adversary by allowing it to "see" all the $(n/2^i + \alpha n \log n)$ random choices made in the phase when choosing the schedule.) To bound the probability that the algorithm fails to move to phase $i+2$, we first *fix* a selection C of $\alpha n \log n$ random choices from this phase, and a set S of less than $n/2^{i+1}$ objects not in D_i. The probability that all the choices in C fall in S or in D_i is at most $\left(1 - 1/2^{i+1}\right)^{\alpha n \log n}$. Using the Bernoulli inequality, we obtain an upper bound of $(1/n)^{\alpha n / 2^{i+1}}$ on this probability. On the other hand, there are at most $2^{n/2^{i+1}}$ possible choices for the set S. Also, there are at most $\binom{n/2^i + \alpha n \log n}{\alpha n \log n}$ ways in which to select the set C. Using the union bound, after some calculation, we obtain that the probability that the algorithm stays in phase $i+1$ after $\alpha n \log n$ accesses is at most $(1/n)^{(\alpha-4)n/2^{i+1}}$.

Since we analyze only the first $\log n$ phases, we obtain that the algorithm moves to phase $i+2$ with high probability for $1 \leq i \leq \log n$. This concludes the induction step for the first part of the claim.

For the second part, let $T_1, T_2, \ldots, T_{n/2^{i+1}}$ be $n/2^{i+1}$ test-and-set objects newly accessed by the algorithm in this phase, which were just shown to exist with high probability. From the properties of test-and-set, it follows that, for every T_j, $1 \leq j \leq n/2^{i+1}$, there exists a process q_j that accesses T_j, but never returns loser from it, i.e. either wins T_j or crashes in T_j. All q_j's must be distinct: a process stops taking steps after winning a test-and-set, and cannot crash in two test-and-sets. Since we consider the accesses in the linearization order, i.e. the winners are the first processes to return from the object, it follows that, with high probability, the processes q_j never take steps in the next phase, as required by the second part of the claim.

To conclude, notice that the second part of the claim proves that all processes return or crash by the end of phase $\log n$. This implies that the algorithm performs a total of $O(n \log^2 n)$ test-and-set accesses, with high probability, before each process terminates. A test-and-set access costs at most $O(\log^2 n)$ steps per process, since repeated accesses by the same process do not add to the complexity of the object. We obtain that the total step complexity is $O(n \log^4 n)$, with high probability. This concludes the proof of Theorem 1. □

6 A Randomized Adaptive Algorithm

In this section, we present a new adaptive randomized renaming algorithm, which we call AdaptiveSearch. Given a constant $\epsilon > 0$, the algorithm guarantees unique names, a namespace of size $\min(k(1+\epsilon), n)$ with high probability, and has $O(k \log^4 k / \log^2(1+\epsilon))$ total step complexity, with high probability.

6.1 The AdaptiveSearch Algorithm

As in the previous algorithm, each process p attempts to choose a name from a vector of n test-and-set objects. We assume that processes share n adaptive test-and-set objects, implemented through the RatRace algorithm, which are numbered from left to right. Since the contention is not known, each process starts with an estimate k_{est} of contention, initially 1, which is increased as needed. Computation proceeds in local phases. In a phase, process p tries to win a randomly chosen test-and-set between 1 and k_{est} for $3 \log k_{est} / \log(1+\epsilon/4)$ times. If it does not succeed by the end of a phase, then the process multiplies k_{est} by a constant factor $(1+\epsilon/4) > 1$, and tries again in the next iteration. Once it succeeds in winning a test-and-set, the process takes the name associated with that test-and-set and returns. We enforce the name returned to be within a namespace of 1 to n as follows: once a process detects that k_{est} is larger than n, it starts to run the ReShuffle algorithm on the n test-and-set instances.

6.2 Analysis of AdaptiveSearch.

In this section, we prove the correctness of AdaptiveSearch and its performance guarantees. Note that name uniqueness is ensured since no two processes may win the same

test-and-set object. Also, AdaptiveSearch ensures termination with probability 1, since we run ReShuffle as a backup. Let k be the contention in the current execution. In this analysis, we assume that the namespace parameter ϵ is less than two (a similar argument holds for $\epsilon \geq 2$).
The first lemma provides an upper bound on the generated namespace with high probability.

Lemma 7 (Namespace). AdaptiveSearch *solves renaming in a namespace from* 1 *to* $k(1+\epsilon)$*, with high probability. The maximum size of the namespace is* n*.*

Proof (Sketch). We consider a process p that obtains a name larger than $k(1+\epsilon)$, and show that the probability that this occurs is very low. First, note that p's estimate of contention k_{est} when it obtained the name must have been at least $k(1+\epsilon)$. Let k_ℓ be the last estimate on contention that process p tried which had the property that $k_\ell < k$. By definition, it follows that $k/(1+\epsilon/4) \leq k_\ell < k$. Since $\epsilon < 2$, we obtain that process p tried to obtain a random name in a namespace of size at least $1, 2, \ldots, k(1+\epsilon/4)$ for at least $3\log k(1+\epsilon/4)/\log(1+\epsilon/4)$ times, and did not succeed.

Next, we notice that, since at most k processes participate in the algorithm, there are at least $k\epsilon/4$ test-and-set objects that are not accessed throughout the entire execution. This follows since, for any test-and-set object that is accessed, there exists at least one process that either wins it or crashes while executing the object, and a process stops taking steps once it acquired a test-and-set. Therefore, irrespective of the adversarial schedule, each process has probability at least $(\epsilon/4)/(1+\epsilon/4)$ to access a test-and-set that is not accessed by another process in the current execution. Also, in this case, the process stops accessing new test-and-set objects. We bound the probability that process p fails to acquire a name within a namespace of size at least $k\epsilon/4+1$ after $3\log(k(1+\epsilon/4))/\log(1+\epsilon/4)$ independent trials. After some calculation, we obtain that this probability is at most $(1/k)^3$. Therefore, with high probability, every process chooses a name between 1 and $k(1+\epsilon)$. □

The next result provides an upper bound on the total step complexity of the algorithm.

Lemma 8 (Complexity). *The* AdaptiveSearch *algorithm takes* $O(k\log^4 k/\log^2(1+\epsilon/4))$ *total steps with high probability.*

Proof. We analyze the total number of steps performed by a process p. By Lemma 7, every process runs for at most $\log_{1+\epsilon/4} k(1+\epsilon)$ local phases, with high probability. In each of these phases, the process performs at most $O(\log k(1+\epsilon)/\log(1+\epsilon/4))$ test-and-set accesses. In turn, each test-and-set is accessed by at most $O(k)$ distinct processes, which implies that one test-and-set access, implemented using the RatRace algorithm, will cost $O(\log^2 k)$ steps per process, with high probability. We thus obtain that the total step complexity is bounded by $O(k\log^4 k/\log^2(1+\epsilon/4))$. □

7 Future work

Our algorithms outline a new approach for solving renaming efficiently in an asynchronous system. One direction for future work is to improve the local (per-process)

step complexity of our algorithms, which may be super-linear in some executions of the ReShuffle algorithm (AdaptiveSearch has poly-logarithmic local step complexity, with high probability). This, together with a multiple-use version of RatRace, would allow our algorithms to be turned into efficient *long-lived* renaming algorithms. Another direction would be to study the lower bounds on the complexity of randomized renaming—we suspect that the lower bound threshold for total step complexity is super-linear. A third direction would be to study whether *tight adaptive* renaming can be achieved efficiently using randomization. It could also be interesting to study whether our approach may be applied to obtain efficient solutions to the well-known Do-All and Write-All (e.g., [17, 18, 22]) problems.

References

1. Yehuda Afek, Eli Gafni, John Tromp, and Paul M. B. Vitányi. Wait-free test-and-set (extended abstract). In *WDAG '92: Proceedings of the 6th International Workshop on Distributed Algorithms*, pages 85–94, London, UK, 1992. Springer-Verlag.
2. Yehuda Afek and Michael Merritt. Fast, wait-free (2k-1)-renaming. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 105–112, New York, NY, USA, 1999. ACM.
3. James H. Anderson and Yong-Jik Kim. Adaptive mutual exclusion with local spinning. In *DISC '00: Proceedings of the 14th International Conference on Distributed Computing*, pages 29–43, London, UK, 2000. Springer-Verlag.
4. James H. Anderson and Mark Moir. Using local-spin k-exclusion algorithms to improve wait-free object implementations. *Distrib. Comput.*, 11(1):1–20, 1997.
5. James Aspnes and Orli Waarts. Randomized consensus in expected $o(nlog^2n)$ operations per processor. *SIAM J. Comput.*, 25(5):1024–1044, 1996.
6. H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, 1990.
7. Hagit Attiya and Keren Censor. Tight bounds for asynchronous randomized consensus. *J. ACM*, 55(5):1–26, 2008.
8. Hagit Attiya and Arie Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31(2):642–664, 2001.
9. Hagit Attiya, Fabian Kuhn, C. Greg Plaxton, Mirjam Wattenhofer, and Roger Wattenhofer. Efficient adaptive collect using randomization. *Distrib. Comput.*, 18(3):179–188, 2006.
10. Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *PODC '93: Proceedings of the twelfth annual ACM symposium on Principles of distributed computing*, pages 41–51, New York, NY, USA, 1993. ACM.
11. Alex Brodsky, Faith Ellen, and Philipp Woelfel. Fully-adaptive algorithms for long-lived renaming. In *DISC*, pages 413–427, 2006.
12. J. E. Burns and G. L. Peterson. The ambiguity of choosing. In *PODC '89: Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 145–157, New York, NY, USA, 1989. ACM.
13. Armando Castañeda and Sergio Rajsbaum. New combinatorial topology upper and lower bounds for renaming. In *PODC '08: Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 295–304, New York, NY, USA, 2008. ACM.
14. Bogdan S. Chlebus and Dariusz R. Kowalski. Asynchronous exclusive selection. In *PODC '08: Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 375–384, New York, NY, USA, 2008. ACM.

15. Wayne Eberly, Lisa Higham, and Jolanta Warpechowska-Gruca. Long-lived, fast, waitfree renaming with optimal name space and high throughput. In *DISC*, pages 149–160, 1998.
16. Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.
17. Chryssis Georgiou, Alexander Russell, and Alexander A. Shvartsman. The complexity of synchronous iterative do-all with crashes. In *DISC*, pages 151–165, 2001.
18. Chryssis Georgiou and Alexander A. Shvartsman. *Do-All Computing in Distributed Systems: Cooperation in the Presence of Adversity*. Springer, 2008.
19. Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, January 1991.
20. Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(2):858–923, 1999.
21. Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
22. Dariusz R. Kowalski and Alexander A. Shvartsman. Writing-all deterministically and optimally using a non-trivial number of asynchronous processors. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 311–320, New York, NY, USA, 2004. ACM.
23. Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York, NY, USA, 2005.
24. Mark Moir and James H. Anderson. Fast, long-lived renaming (extended abstract). In *WDAG '94: Proceedings of the 8th International Workshop on Distributed Algorithms*, pages 141–155, London, UK, 1994. Springer-Verlag.
25. Mark Moir and Juan A. Garay. Fast, long-lived renaming improved and simplified. In *WDAG '96: Proceedings of the 10th International Workshop on Distributed Algorithms*, pages 287–303, London, UK, 1996. Springer-Verlag.
26. Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge University Press, New York, NY, USA, 1995.
27. Alessandro Panconesi, Marina Papatriantafilou, Philippas Tsigas, and Paul M. B. Vitányi. Randomized naming using wait-free shared variables. *Distributed Computing*, 11(3):113–124, 1998.
28. John Tromp and Paul Vitányi. Randomized two-process wait-free test-and-set. *Distrib. Comput.*, 15(3):127–135, 2002.