The Complexity of Renaming

Dan Alistarh
EPFL

James Aspnes
Yale

Seth Gilbert
NUS

Rachid Guerraoui
EPFL

Abstract

We study the complexity of renaming, a fundamental problem in distributed computing in which a set of processes need to pick distinct names from a given namespace. We prove a *local* lower bound of $\Omega(k)$ process steps for deterministic renaming into any namespace of size sub-exponential in k, where k is the number of participants. This bound is tight: it draws an exponential separation between deterministic and randomized solutions, and implies new tight bounds for deterministic fetch-and-increment registers, queues and stacks. The proof of the bound is interesting in its own right, for it relies on the first reduction from renaming to another fundamental problem in distributed computing: mutual exclusion. We complement our local bound with a *global* lower bound of $\Omega(k \log(k/c))$ on the *total* step complexity of renaming into a namespace of size ck, for any $c \geq 1$. This applies to randomized algorithms against a strong adversary, and helps derive new global lower bounds for randomized approximate counter and fetch-and-increment implementations, all tight within logarithmic factors.

1 Introduction

Unique identifiers are a fundamental prerequisite for efficiently solving a variety of problems in distributed systems. In some settings, such as Ethernet networks, unique names are available, but come from a very large namespace, which reduces their usefulness. Thus, the problem of assigning small unique names to a set of participants, known as *renaming* [6], is fundamental in distributed computing.

A lot of work has been devoted to devising renaming algorithms that minimize the size of the available namespace, whether deterministically, e.g. [6, 11, 12, 27, 28] or using randomization [3, 4, 14, 29]. Herlihy and Shavit [21], as well as Rajsbaum and Castañeda [13], showed that deterministic wait-free renaming is impossible in less than $(2k-1)$ names, where k is the number of participants. Algorithms using randomization or atomic compare-and-swap operations are able to circumvent this impossibility and assign a *tight* namespace of k consecutive names to the k participants.

Little is known about the complexity of renaming. Establishing a complexity bound for renaming is challenging, especially when the namespace is *loose* (non-tight): intuitively, the difficulty comes from the fact that the number of possible outputs increases exponentially with the size of the allowed namespace.

In this paper, we study the complexity of renaming. Our study covers both randomized and deterministic algorithms, and is the first to apply to algorithms that assign a loose namespace.

We present a lower bound on the number of process steps (reads, writes, and compare-and-swap operations) for renaming into any namespace of size sub-exponential in the number of participants k (renaming into a namespace whose size depends on the number of participants is also called *adaptive* renaming [8]). We prove that any deterministic algorithm that renames into a namespace of size at most $2^{f(k)}$, for any function $f(k)$ in $o(k)$, has runs in which a process performs $\Omega(k)$ steps. This result is somewhat surprising, since it shows that assigning names adaptively in a huge namespace, e.g. of size $\Theta(k^{100})$, is no easier (asymptotically) than renaming in a small namespace of size $O(k)$. The lower bound holds even in a system with no failures, and even if the devices have access to powerful synchronization primitives like compare-and-swap. In essence, we show that some process must pay a linear cost for "contention resolution," i.e., competing for a name, even if the namespace is extremely sparse. The lower bound is tight both for algorithms that use atomic compare-and-swap operations [27], and for read-write algorithms, e.g. [8, 28]. The bound highlights an exponential complexity separation between deterministic and randomized adaptive renaming, since there exist randomized renaming algorithms with $O(\log k)$ expected local (per-process) step complexity [3].

We believe that the proof of our renaming lower bound is interesting in its own right, since it follows from relating renaming with mutual exclusion. This highlights the first connection between two fundamental distributed computing problems, so far studied independently. More precisely, we start from a renaming algorithm, and use a data structure based on AKS sorting networks [2] to efficiently reduce the size of the namespace. We show that the resulting construction can be used to solve mutual exclusion while preserving the asymptotic step complexity of the original renaming algorithm. The transformation is efficient as long as the original algorithm renames in a namespace of size sub-exponential in k. We then obtain our result from a linear lower bound on the complexity of mutual exclusion [24]. Of note, we prove that this technique implies a similar linear lower bound for the step complexity of *non-adaptive* renaming (where the size of the final namespace is fixed in advance). Another side result of the reduction is a new asymptotically optimal mutual exclusion algorithm based on an AKS sorting network.

More generally, our reduction connects the complexity of wait-free algorithms that rely on implicit synchronization primitives (such as compare-and-swap) with the complexity of locking implementations that rely on explicit synchronization (such as busy-waiting loops). In fact, our reduction technique implies a stronger $\Omega(k)$ lower bound on the number of *remote memory references* (RMRs) [9, 24] that a process has to perform in worst-case executions, which are orders of magnitude slower than accesses to local memory on most multi-processor architectures. Since renaming can be achieved wait-free with $O(k)$ RMRs, this also

Shared Object	Lower Bound	Type	Matching Algorithms	New Result
Deterministic c-loose Renaming	$\Omega(k)$	Local	[8, 27, 28]	Yes
	$\Omega(k \log(k/c))$	Global	[3]	Yes
Randomized c-loose Renaming	$\Omega(k \log(k/c))$	Global	[3]	Yes
Randomized c-approx. Counter	$\Omega(k \log(k/c))$	Global	[5]	Yes
Fetch-and-Increment	$\Omega(k)$	Local	[3, 27]	Improves on [15]
	$\Omega(k \log k)$	Global	[3]	Improves on [7]
Queues and Stacks	$\Omega(k)$	Local	Universal Constructions [1, 20]	Improves on [15]
	$\Omega(k \log k)$	Global	-	Improves on [7]

Figure 1: Summary of results and relation to previous work.

shows that there is no advantage to local spinning (i.e., busy waiting loops) when solving renaming. (By contrast, efficient solutions to mutual exclusion require local spinning.)

Our lower bound has ramifications beyond renaming. Since adaptive renaming can be easily solved using either a fetch-and-increment register, or an initialized queue or stack, our lower bound applies to these objects as well. We obtain new lower bounds for fetch-and-increment, queues and stacks, which improve on previously known results [15,23]. In particular, our result suggests that the cost of implementing shared objects such as queues is at least as high as the cost of implementing locks, even when implicit synchronization primitives are available. Since we count RMRs, and only require one operation per process in worst-case executions (as opposed to exponentially many), our result is stronger than those of [15, 23].

We complement our local step complexity[1] lower bound by also analyzing the *total* number of steps that processes perform in a worst-case execution. We prove a *global* step complexity lower bound for randomized adaptive renaming against a strong adversary: given any algorithm that renames into a namespace of size ck with $c \geq 1$, there exists an adversarial strategy that causes the processes to take $\Omega(k \log(k/c))$ total steps in expectation. For this, we employ an information-theoretic technique to bound the knowledge that a process may gather throughout an execution: we start from an adversarial strategy which forces *each* process to take $\Omega(\log \ell)$ steps to find out about ℓ other participating processes, and prove that, roughly, a process returning name ℓ has to know that at least ℓ/c other processes have taken steps. Since the names returned have to be distinct, the lower bound follows. This total step complexity lower bound is tight for $c = 1$, i.e. for *strong* adaptive renaming, since it is matched by the randomized algorithm of [3]. It is also tight within logarithmic factors for deterministic algorithms that use atomic compare-and-swap operations [3].

This global technique implies new total step complexity lower bounds for randomized implementations of approximate counters, fetch-and-increment registers, queues, and stacks. (The results are summarized in Figure 1.) The technique improves on previous lower bounds for these objects [7, 9, 22], since it covers the *global* complexity of *randomized approximate* solutions. Our global results are tight within logarithmic factors for counters [5] and fetch-and-increment registers [3]. Since the lower bounds apply to randomized algorithms, and the almost-matching algorithms [3, 5] are deterministic, this suggests that, against a strong adversary, the complexity improvement that may be obtained through randomization can be at most logarithmic. (For approximate counters, the lower bound also limits the complexity gain from allowing approximation within constant factors.) This global technique also applies to randomized algorithms that may not terminate with some non-zero probability.

Roadmap. The model and problem statements are defined in Section 2. We prove the main lower bound in Section 3, and the global lower bound in Section 4. Section 5 presents the ramifications to other shared objects. Section 6 presents an overview of related work, while Section 7 summarizes our results. Due to space limitations, we present proof sketches for some claims; the full proofs can be found in the Appendix.

[1] By *local* step complexity we mean the number of per-process shared memory operations, not the local computation steps.

2 Preliminaries

Assumptions. We consider a standard asynchronous shared memory model with n processes, $t < n$ of which may fail by crashing, also called the *wait-free* model. Each process has a *unique* initial identifier p_i, from a namespace of unbounded size. We consider k to denote total *contention*, i.e. the total number of processes that take steps during a certain execution. Processes may know n, but do not know k. Processes communicate through multiple-writer-multiple-reader atomic registers. Any register R exports atomic read, write, and compare-and-swap operations, with the usual semantics. The *test-and-set* object has initial value 0, and exports an atomic test-and-set operation, which atomically reads the value and sets it to 1 (compare-and-swap can simulate test-and-set at no extra cost).

Process failures and scheduling are controlled by an adaptive adversary (also called a *strong* adversary). For randomized algorithms, at any point in the execution the adversary knows the results of the random coin flips that the processes have performed, and can adjust the schedule and the failure pattern accordingly.

Problem Statement. The *renaming* problem [6] requires each correct (non-faulty) process to eventually return a name, and that the names returned should be unique. For *adaptive* renaming, the size of the resulting namespace should only depend on the number of participants k (as opposed to n for *non-adaptive* renaming). The adaptive *strong* renaming problem requires the size of the namespace to be exactly k. The c-*loose* renaming problem requires names to be between 1 and ck, for any constant $c \geq 1$. The *counter* object exports operations increment and read with the same semantics as the sequential object. A c-approximate counter implementation has the property that the result returned by a read operation is always between v/c and $c \cdot v$, where v is the number of increment operations linearized before it.

The *mutual exclusion* object supports two operations enter and exit. The enter operation allows the process to enter the critical section; after competing the critical section, a process invokes the exit operation. A correct mutex implementation satisfies (1) *mutual exclusion*: at most one process can be in the critical section at any given point in time; (2) *deadlock freedom*: if some process calls enter, then, eventually, a process enters the critical section; (3) *finite exit*: every process completes exit in a finite number of steps. The *adaptive* mutual exclusion problem requires the complexity of the algorithm to depend on the number of participants k (instead of n); the correctness conditions above remain unchanged.

Complexity Measures. We measure complexity in terms of process steps: each shared-memory operation is counted as one step, coin flips are not counted. The *total step complexity* is the total number of process shared-memory operations in an execution, while *local step complexity* (or simply *step complexity*) is the number of shared-memory operations a single process may perform during an execution.

For the bound in Section 3, we provide a stronger measure of complexity by counting remote memory references (RMRs) [9, 24]. In cache-coherent (CC) shared memory, each processor maintains local copies of shared variables inside its cache, whose consistency is ensured by a coherence protocol. A variable is *remote* to a processor if its cache contains a copy of the variable that is out of date; otherwise, the variable is *local*. A step is *local* if it accesses a local variable; otherwise it is a *remote memory reference* (RMR). A similar definition exists for the distributed shared memory (DSM) model. For a more precise description of RMRs, please see [9, 24]. For wait-free algorithms such as the renaming algorithms we consider, which do not employ busy-waiting loops, the RMR complexity is always a lower bound on their step complexity.

3 The Main Lower Bound

In this section, we prove an $\Omega(k)$ lower bound on the step and RMR complexity of any deterministic adaptive renaming algorithm that renames into a namespace of size sub-exponential in k. We believe the technique is interesting in its own right, as it establishes a reduction between renaming and mutual exclusion using

optimal-depth AKS sorting networks [2] as an intermediate step. Due to space limitations, we present detailed proof sketches for some of the claims; the complete proof can be found in the Appendix.

Theorem 1 (Local Lower Bound)**.** *For any $k \geq 1$, any wait-free deterministic adaptive renaming algorithm which renames into a namespace of size at most $2^{f(k)}$ for any function $f(k) = o(k)$ has a worst-case execution $\mathcal{E}$ in which (1) some process performs $\Omega(k)$ RMRs (and $\Omega(k)$ steps) and (2) each participating process performs a single* rename *operation.*

Proof. We begin by assuming for contradiction that there exists a deterministic adaptive algorithm R that renames into a namespace of size $M(k) = 2^{f(k)}$ for $f(k) \in o(k)$, with step complexity $C(k) = o(k)$. The first step in the proof is to show that any such algorithm can be transformed into a wait-free algorithm that solves adaptive *strong* renaming in the same model; the complexity cost of the resulting algorithm will be $O(C(k) + \log M(k))$.

Claim 1. *Any wait-free algorithm R that renames into a namespace of size $M(k)$ with complexity $C(k)$ can be transformed into a tight adaptive renaming algorithm $T(R)$ with complexity $O(C(k) + \log M(k))$.*

Proof (Sketch). We start from a recursive data structure based on AKS sorting networks [2] where the comparators are replaced with test-and-set objects, called an *adaptive renaming network* [3]. An adaptive renaming network is a construction with an unbounded number of ports, which ensures the properties of a sorting network whenever truncated to any number of input (and output) ports; every comparator in the original sorting network is replaced with a (two-process) test-and-set object.

Given an AKS adaptive renaming network A, we use the algorithm R to assign unique input ports to processes in the renaming network. More precisely, a process first calls the algorithm R to obtain a temporary name; the process then uses this name as an input port for the adaptive AKS renaming network. Once it has an input port by returning from R, the process starts at that port and follows a path through the network determined by leaving each comparator on its lower output wire, if it wins the test-and-set (i.e. returns 0 from the test-and-set invocation), and on the higher output wire otherwise (wires are indexed from top to bottom); the process returns the index of its output port as its final name in the algorithm $T(R)$.

Starting from the properties of renaming networks [3], we show that the algorithm $T(R)$ has the following properties: (1) every correct process eventually reaches a unique output port; (2) in every execution, the k participants may exit the renaming network only on the first (highest) k output ports of the network; (3) if $M(k)$ is the size of the namespace generated by algorithm R and O is the largest index of an occupied output port, then the largest number of steps a process performs in the renaming network is $O(\log \max(M(k), O))$.

Properties (1) and (2) ensure that the composition of R and A solves strong adaptive renaming. Properties (2) and (3) imply that the complexity of the resulting algorithm is $O(C(k) + \log \max(M(k), k)) = O(C(k) + \log M(k))$. □

Returning to the main proof, in the context of algorithm R, the claim guarantees that the algorithm $T(R)$ solves strong adaptive renaming with complexity $o(k) + O(\log 2^{f(k)}) = o(k) + O(f(k)) = o(k)$.

The second step in the proof shows that any wait-free strong adaptive renaming algorithm can be used to solve adaptive mutual exclusion with only a constant blowup in terms of complexity.

Claim 2. *Any deterministic algorithm R for adaptive strong renaming implies a correct adaptive mutual exclusion algorithm $ME(R)$. The RMR complexity of $ME(R)$ is upper bounded asymptotically by the RMR complexity of R, which is in turn upper bounded by its step complexity.*

Proof (Sketch). First, note that the models in which renaming and mutual exclusion are generally considered are distinct: renaming assumes a wait-free shared-memory model WF which may be augmented with atomic compare-and-swap or test-and-set operations; complexity is measured in the number of steps that a process

takes during the execution. Mutual exclusion assumes a more specific cache-coherent (CC) or distributed shared memory (DSM) shared-memory model with no process failures (otherwise, a process crashing in the critical section could block the processes in the entry section forever); the complexity of mutex algorithms is measured in terms of remote memory references (RMRs). We call this second model the *failure-free, local spinning* model, in short LS.

The transformation from adaptive tight renaming algorithm R in WF to the mutex algorithm $ME(R)$ in LS is as follows. We assume that processes share algorithm R, and a vector $Done$ of boolean bits, initially set to false. In the first stage of the transformation, we consider the subset of registers used by the algorithm R on which either compare-and-swap or test-and-set operations are performed. We replace every such register with a read-write compare-and-swap implementation with constant RMR complexity using the techniques presented in [18, 19]. (Note that this compare-and-swap implementation also supports atomic read and write operations.) We obtained an adaptive strong renaming algorithm R' in the LS model whose RMR complexity is the same (modulo constant factors) as the step complexity of R.

We now solve adaptive mutual exclusion using algorithm R'. For the enter procedure of the mutex implementation, each of the k participating processes runs algorithm R', and obtains a unique name from 1 to k. The process that obtained name 1 enters the critical section; upon leaving, it sets the $Done[1]$ bit to true. Any process that obtains a name $id \geq 2$ from the adaptive renaming object spins on the $Done[id-1]$ bit associated to name $id-1$, until the bit is set to true. When this occurs, the process enters the critical section. When calling the exit procedure to release the critical section, each process sets the $Done[id]$ bit associated with its name to true and returns.

We prove that algorithm $ME(R)$ is a correct mutex implementation using the tightness and adaptivity of the namespace generated by algorithm R (please see the Appendix for a full proof). For the complexity claims, first notice that the RMR complexity of the algorithm R' is at most a constant times the RMR complexity of algorithm R. Next, notice that, once a process obtains the name from algorithm R', it performs exactly one extra RMR before entering the critical section, since an RMR is charged only when it sees the true value in the $Done[v-1]$ register, which may occur at most once. Therefore, the (local or global) RMR complexity of the mutex algorithm is at most a constant times the (local or global) RMR complexity of the original algorithm R. Since the algorithm R is wait-free, therefore uses no busy-waiting loops, its RMR complexity is a lower bound on its step complexity, which concludes the proof of the claim. □

To conclude the proof of Theorem 1, notice that the algorithm resulting from the composition of the two claims, $ME(T(R))$, is an adaptive mutual exclusion algorithm with RMR complexity $o(k) + O(f(k)) = o(k)$. However, the existence of this algorithm contradicts the $\Omega(k)$ lower bound on the RMR complexity of adaptive mutual exclusion by Anderson and Kim [24]. The contradiction arises from our initial assumption on the existence of algorithm R. The claim about step complexity follows since, for wait-free algorithms, the RMR complexity is always a lower bound on step complexity. The claim about the number of rename operations follows from the structure of the transformation and from that of the mutual exclusion lower bound of [24]. □

Relation between k and n. The lower bound of Anderson and Kim [24] from which we obtain our result assumes large values of n, the maximum possible number of participating processes, in the order of k^{2^k}. Therefore, algorithms that optimize for smaller values of n may be able to circumvent the lower bound for particular combinations of n and k. On the other hand, the lower bound applies to all algorithms that work for arbitrary values of n and k.

Non-Adaptive Renaming. This technique also implies an $\Omega(n)$ lower bound on the step complexity of non-adaptive renaming algorithms (for which the size of the resulting namespace depends on n, not on k). The result follows from a new reduction from non-adaptive renaming to adaptive renaming, which preserves asymptotic step complexity and polynomial namespace size. The proof can be found in the Appendix.

```
1  procedure adversarial-scheduler()
2  r ← 1
3  while true do
4      for each process p do
5          schedule p to perform coin flips until it has enabled a shared-memory operation, or p returns
6      R ← processes that have read operations enabled
7      W ← processes that have write operations enabled
8      C ← processes that have compare-and-swap operations enabled
9      schedule all processes in R to perform their operations, in the order of their initial identifiers
10     schedule all processes in W to perform their operations, in the order of their initial identifiers
11     schedule all processes in C to perform their in the order defined by the secretive schedule σ
12     r ← r + 1
```

Algorithm 1: The adversarial strategy for the global lower bound.

Corollary 1. *For any $n \geq 1$, any wait-free deterministic (non-adaptive) renaming algorithm that renames into a polynomial namespace has an execution in which a process performs $\Omega(n)$ steps.*

An Optimal Non-Adaptive Mutex Protocol. Another application of the lower bound argument is that we can obtain an asymptotically optimal mutual exclusion algorithm from an AKS sorting network [2].

Processes share an AKS sorting network with n input (and output) ports, and a vector $Done$ of boolean bits, initially false. We replace each comparator in the network with a two-process test-and-set object with constant RMR complexity [18,19]. In the mutual exclusion problem processes hold unique initial identifiers from 1 to n, therefore we use these initial identifiers to assign unique input ports to processes. A process progresses through the network starting at its input port, competing in test-and-set objects. A process takes the lower comparator output if it wins (returns 0 from) the test-and-set, and the higher output otherwise. The process adopts the index of the output port it reaches as a temporary name id. If $id = 1$, then it enters the critical section; otherwise it busy-waits until the bit $Done[id - 1]$ is set to true. Upon exiting the critical section, the process sets the $Done[id]$ bit to true.

The correctness of the algorithm above follows from Claims 1 and 2. In particular, the asymptotic local RMR complexity of the above algorithm is the same as the depth of the AKS sorting network (plus one RMR), i.e. $O(\log n)$, therefore the algorithm is optimal by the lower bound of Attiya et al. [9]. Anderson and Yang [30] presented an upper bound with the same asymptotic complexity, but better constants, using a different technique. The same construction can be used starting from constuctible sorting networks, e.g. bitonic sorting networks [25], at the cost of increased complexity.

4 The Total Step Complexity Lower Bound

In this section, we present lower bounds on the *total* step complexity of randomized renaming and counting. We start from an adversarial strategy that schedules the processes in lock-step, and show that this limits the amount of information that each process may gather throughout an execution. We then relate the amount of information that *each* process must gather with the set of names that the process may return in an execution. For executions in which everyone terminates and the adversary follows the lock-step strategy, we obtain a lower bound of $\Omega(k \log(k/c))$ for c-loose renaming. We then notice that a similar argument can be applied to obtain a lower bound for c-approximate counting.

Strategy Description. We consider an algorithm A in shared-memory augmented with atomic compare-and-swap operations. The adaptive adversary follows the steps described in Algorithm 1. The adversary

schedules the processes in rounds: in each round, each process that has not yet returned from A is scheduled to perform a shared-memory operation. More precisely, at the beginning of each round, the adversary allows each process to perform local coin flips until it either terminates or has to perform an operation that is either a read, a write, or a compare-and-swap (lines 3-5).

The adversary partitions processes into three sets: $\mathcal{R}$, the *readers*, $\mathcal{W}$, the *writers*, and $\mathcal{C}$, the *swappers*. Processes in $\mathcal{R}$ are scheduled by the adversary to perform their enabled read operations, in the order of their initial identifiers (line 9). Then each process in $\mathcal{W}$ is scheduled to perform the write, again in the order of initial identifiers (line 10). Finally, the swappers are scheduled following a particular *secretive* schedule σ, defined in Lemma 1, whose goal is to minimize the information flow between processes. Once each process has either been scheduled or has returned, the adversary moves on to the next round.

Before we proceed with the analysis, we define the schedule for the processes performing compare-and-swap operations in round r. Notice that, if a set of processes all perform compare-and-swap operations in a round, there exist interleavings of these operations such that the last scheduled process finds out about *all* other processes after performing its compare-and-swap. However, the adversary can always break such interleavings and ensure that, given any set of compare-and-swap operations, a process only finds out about a constant number of other processes, using a *secretive* schedule, introduced in [22]. We re-state the definition and properties of secretive schedules [22] in our model.

Lemma 1 (Secretive Schedules [22])**.** *Given a set of* compare-and-swap *operations enabled after some execution prefix $\mathcal{P}$, there exists an ordering σ of these events, called* secretive*, such that the value originating at any process reaches at most two other processes.*

Analysis. First, notice that, since the algorithms we consider are randomized, the adversarial strategy we describe creates a set of executions in which all processes take steps (if the algorithm is deterministic, then the strategy describes a single execution). We denote the set of such executions by $\mathcal{S}(A)$. In the following, we study the flow of information between the processes in executions from $\mathcal{S}(A)$.

We prove that the adversarial strategy described above prevents any process from "finding out" about more than 4^r active processes by the end of round r in any execution from $\mathcal{S}(A)$. In particular, for each process p following the algorithm A, each register R, and for every round $r \geq 0$, we define the sets $UP(p, r)$ and $UP(R, r)$, respectively. Intuitively, $UP(p, r)$ is the set of processes that process p might know at the end of round r as having taken a step in an execution resulting from the adversarial strategy. Similarly, $UP(R, r)$ is the set of processes that can be inferred to have taken a step in an execution resulting from the adversarial strategy, at the end of round r. Our notation follows the one in [22], which defines similar measures for a model in which LL/SC, move, and swap operations are available.

Formally, we define these sets inductively, using the following update rules. Initially, for $r = 0$, we consider that $UP(p, 0) = \{p\}$ and $UP(R, 0) = \emptyset$, for all processes p and registers R. For any later round $r \geq 1$, we define the following update rules: (1) At the beginning of round $r \geq 1$, for each process p and register R, we set $UP(p, r) = UP(p, r-1)$ and $UP(R, r) = UP(R, r-1)$; (2) If process p performs a successful write operation on register R in round r, then $UP(R, r) = UP(p, r-1)$. Informally, the knowledge that process p had at the end of round $r-1$ is reflected in the contents of register R at the end of round r. On the other hand, the writing process p gains no new knowledge from writing, i.e. $UP(p, r) = UP(p, r-1)$; (3) If process p performs a successful compare-and-swap operation on register R in round r, i.e. if the operation returns the expected value, then the information contained in the register is overwritten with p's information, that is $UP(R, r) = UP(p, r-1)$. We also assume that the process p gets the information previously contained in the register $UP(p, r) = UP(p, r-1) \cup UP(R, r)$ (the contents of $UP(R, r)$ might have been already updated in round r); (4) If process p performs an unsuccessful compare-and-swap operation on register R in round R, then $UP(R, r)$ remains unchanged. On the other hand, the process gets the information currently contained in the register, i.e. $UP(p, r) = UP(p, r-1) \cup$

$UP(R, r)$; (5) If process p performs a successful read operation on register R in round r, then $UP(R, r)$ remains unchanged, and $UP(p, r) = UP(R, r) \cup UP(p, r-1)$.
Based on these update rules, we can easily compute an upper bound on the size of the UP sets for processes and registers. The proof follows by induction on the round number $r \geq 0$.

Lemma 2 (Bounding information)**.** *Given a run of the algorithm A controlled by the adversarial scheduler in Algorithm 1, for any round $r \geq 0$, and for every process or shared register or compare-and-swap X, $|UP(X, r)| \leq 4^r$.*

Indistinguishability. Let $\mathcal{E}$ be an execution of the algorithm obtained using the adversarial strategy above, i.e. $\mathcal{E} \in \mathcal{S}(A)$. Given the previous construction, the intuition is that, for a process p and a round r, if $UP(p, r) = S$ for some set S, then p has no evidence that any process outside the set S has taken a step in the current execution $\mathcal{E}$. Alternatively, there exists a parallel execution $\mathcal{E}'$ in which only processes in the set S take steps, and p cannot distinguish between the two executions.

We make this intuition precise. First, we define $\mathsf{state}(\mathcal{E}, p, r)$ as the local state of process p at the end of round r (i.e. the values of its local registers and its current program counter), and $\mathsf{val}(\mathcal{E}, R, r)$ as the value of register R at the end of round r. We also define $\mathsf{numtosses}(\mathcal{E}, p, r)$ as the number of coin tosses that the process p performed by the end of round r of $\mathcal{E}$. Two executions $\mathcal{E}$ and $\mathcal{E}'$ are said to be *indistinguishable* to process p at the end of round r if (1) $\mathsf{state}(\mathcal{E}, p, r) = \mathsf{state}(\mathcal{E}', p, r)$, and (2) $\mathsf{numtosses}(\mathcal{E}, p, r) = \mathsf{numtosses}(\mathcal{E}', p, r)$.

Starting from the execution $\mathcal{E}$, the adversary can build an execution $\mathcal{E}'$ in which only processes in S participate, that is indistinguishable from $\mathcal{E}$ from p's point of view, by starting from execution $\mathcal{E}$ and only scheduling processes in $S = UP(p, r)$ up to the end of round r of $\mathcal{E}'$. The proof is similar to one presented by Jayanti [22] in the context of local lower bounds in the LL/SC model. Therefore, we only present an outline of the construction, which can be found in the Appendix.

Lemma 3 (Indistinguishability)**.** *Let $\mathcal{E}$ be an execution in $\mathcal{S}(A)$ and p be a process with $UP(p, r) = S$ at the end of round r. There exists an execution $\mathcal{E}'$ of A in which only processes in S take steps, such that $\mathcal{E}$ and $\mathcal{E}'$ are* indistinguishable *to process p.*

Renaming Lower Bound. We now prove an $\Omega(k \log(k/c))$ lower bound on the total step complexity of c-loose adaptive renaming algorithms. In particular, this lower bound implies that we cannot gain more than a constant factor in terms of step complexity by relaxing the tight namespace requirement by a constant factor. There are two key technical points: first, we relate the amount of information that a process gathers with the set of names it may return (we show this relation holds even if renaming is loose); second, for each process, we relate the number of steps it has taken with the amount of information it has gathered.

Theorem 2 (Renaming)**.** *Fix $c \geq 1$ constant. Given k participating processes, any c-loose adaptive renaming algorithm that terminates with probability α has worst-case expected total step complexity $\Omega(\alpha k \log(k/c))$.*

Proof. Let A be a c-loose adaptive renaming algorithm. We consider a *terminating* execution $\mathcal{E} \in \mathcal{S}(A)$ with k participating processes, i.e. every participating process returns in $\mathcal{E}$. We first prove that a process that returns name $j \in [1, ck]$ in execution $\mathcal{E}$ has to perform $\Omega(\log(j/c))$ shared-memory operations. First, notice that each execution $\mathcal{E} \in \mathcal{S}(A)$ contains no process failures, so each process has to return a unique name in the interval $1, \ldots, ck$ in such an execution. Therefore, there exist distinct names $m_1, \ldots, m_k \in \{1, 2, \ldots, ck\}$ and processes $q_1, \ldots, q_k$ such that process q_i returns name m_i in execution $\mathcal{E}$. W.l.o.g, assume that the names are in increasing order; since they are distinct, we have that $m_i \geq i$ for $i \in 1, \ldots, k$.

Consider process q_i returning name m_i in $\mathcal{E}$. Let ℓ_i be the number of shared-memory operations that q_i has performed in $\mathcal{E}$. Since the adversary schedules each process once in every round of $\mathcal{E}$, until termination,

it follows that process q_i has returned at the end of round ℓ_i. Let $S = UP(q_i, \ell_i)$. Since $\mathcal{E} \in \mathcal{S}(A)$, by Lemma 2, we have that $|S| \leq 4^{\ell_i}$.

Assume for the sake of contradiction that the number of processes that q_i found out about in this execution, $|S|$, is less than m_i/c. By Lemma 3, there exists an execution $\mathcal{E}'$ of A which is indistinguishable from $\mathcal{E}$ from q_i's point of view at the end of round ℓ_i, in which only $|S| < m_i/c$ processes take steps. However, since the algorithm is c-loose, the highest name that process q_i can return in execution $\mathcal{E}'$, and thus in $\mathcal{E}$, is *strictly less* than $c \cdot (m_i/c) = m_i$, a contradiction.

From the previous argument, we obtain that $|S| \geq m_i/c$, which implies that ℓ_i, the number of shared-memory operations that process q_i performs in $\mathcal{E}$, is at least $\frac{1}{2}\log(m_i/c)$. Therefore, for any $i \in 1, \ldots, k$, process q_i returning name m_i has to perform at least $\frac{1}{2}\log(m_i/c)$ shared memory operations. Then the total number of steps that the processes perform in $\mathcal{E}$ is $\frac{1}{2}\sum_{i=1}^{k} \ell_i \geq \frac{1}{2}\sum_{i=1}^{k} \log(i/c) = \Omega(k \log(k/c))$.
Since this complexity lower bound holds for every execution resulting from the adversarial strategy, we obtain that the expected total step complexity of the algorithm A is $\Omega(\alpha k \log(k/c))$. □

Counting Lower Bound. Using a similar argument, we can show that any c-approximate counter implementation has worst-case expected total step complexity $\Omega(k \log(k/c^2))$ in executions where each process performs one increment and one read. Since the almost-matching algorithm [5] is deterministic and exact, this bound limits the gain that can be obtained by randomization or approximation to a constant factor.

One key difference in the proof (which implies the extra c factor) is that processes may return the same value from the read operation; we take this into account by studying the linearization order of the increment operations. The proof can be found in the Appendix.

Theorem 3 (Counting)**.** *Fix $c \geq 1$ constant. Let A be a linearizable c-approximate counter implementation that terminates with probability α. For any k, the algorithm A has worst-case expected total step complexity $\Omega(\alpha k \log(k/c^2))$, in runs where each process performs an* increment *followed by a* read *operation.*

5 Ramifications

We find that our results imply local and global lower bounds for implementations of other shared objects, such as fetch-and-increment registers, queues, and stacks. Some of these results are new, while others improve on previously known results.

We first show reductions between fetch-and-increment, queues, and stacks, on the one hand, and adaptive strong renaming, on the other hand. Given a linearizable fetch-and-increment register, we can trivially solve adaptive strong renaming by having each participant call the fetch-and-increment operation once, and return the value received plus 1. Given a linearizable shared queue initialized with n distinct objects $1, 2, \ldots, n$, we can solve adaptive strong renaming by having each participant call the dequeue operation once, and return the value received. The transformation is similar for a stack. (These reductions are formalized in the Appendix.) This implies local and global lower bounds for these objects.

Corollary 2 (Applications)**.** *Consider a wait-free linearizable implementation A of a fetch-and-increment register, queue, or stack, in shared memory with compare-and-swap operations. The following hold.*

- *If the algorithm A is deterministic, then, for any k, there exists an execution of A with k participants in which (1) each participant performs a single operation, and (2) some process performs $\Omega(k)$ RMRs.*
- *If the algorithm A is randomized, then, for any k, if A terminates with probability α, then its expected worst-case step complexity is $\Omega(\alpha k \log k)$, where k is the number of participating processes.*

6 Related Work

Renaming was introduced in [6], where the authors proposed a wait-free solution using $(2n-1)$ names in an asynchronous system, and showed that at least $(n+1)$ names are required in the wait-free case. The lower bound on the size of the namespace for deterministic read-write solutions was improved to $(2n-2)$ in a landmark paper by Herlihy and Shavit [21], with refinements by Rajsbaum and Castañeda [13]. This lower bound can be circumvented using hardware compare-and-swap or test-and-set operations [27], as well as using randomization (at the cost of allowing a vanishing probability that the algorithm does not terminate). Renaming has been shown to be related to set agreement and to weak symmetry breaking in [16, 17]; it is also related to the processor identity problem [26]; the key difference is that, for renaming, participants are assumed to have distinct initial identifiers (from an unbounded namespace).

Several renaming algorithms were proposed in the literature, e.g. [3, 4, 6, 8, 11, 27, 28]. The randomized renaming algorithm of [3] is optimal, as per our global lower bound. On the other hand, our local lower bound is matched by the algorithm of [27] assuming compared-and-swap operations; for read-write algorithms, it is matched by [8, 28]. The only known optimality result was a logarithmic lower bound on the local step complexity of *strong* renaming, derived in [3] using a technique by Jayanti [22]. Our lower bound generalizes Jayanti's result [22] in two ways: first, since we consider *total* step complexity, our results imply the local bounds of [22]; second, we consider the *loose* version of the problem, which relaxes the tight namespace requirements.

As we pointed out, our local lower bound applies to counters, fetch-and-increment, queues, and stacks, and extends previous results obtained on these objects. Indeed, Jayanti, Tan and Toueg [23], as well as Ellen et al. [15], already presented linear lower bounds for deterministic counters, queues and stacks. One limitation of these two results is that the worst-case executions they build require processes to perform an exponential number of operations–by contrast, there exist counter implementations that have polylogarithmic step complexity for polynomially many increment operations [5]. Our linear local bound does not have this limitation, since each process performs only one operation in the worst-case execution. In essence, we show that the linear threshold is inherent for worst-case executions of fetch-and-increment, queues, and stacks, even if each process performs only one operation. Our global lower bound is the first to cover randomized and approximate implementations and in this sense generalizes the lower bounds of Attiya et al. [7, 9].

7 Summary and Future Work

We prove tight bounds for assigning unique names using a shared memory. We cover the local and global cost of adaptive renaming, both for deterministic and randomized solutions. In short, we prove a linear per-process cost to deterministic renaming, which cannot be overcome as long as the name space is sub-exponential, and a logarithmic average local cost, which cannot be avoided using randomization or relaxing the namespace size by a constant factor. Our results imply new tight bounds for counters, queues, and stacks.

One way to circumvent our lower bounds would be to assume a weaker adversary, or to allow some probability of error for the algorithms. In particular, it is known [10] that approximate counting can be achieved with $O(\log \log n)$ expected steps against an oblivious adversary, that fixes the schedule in advance.

Our quest towards these lower bounds revealed the first connection between renaming and another fundamental problem in distributed computing: mutual exclusion. This connection opens the possibility of deriving new lower bounds on randomized mutual exclusion from renaming lower bounds. We also highlighted the central role of sorting networks when reasoning about both the mutual exclusion and renaming problems. Precisely characterizing this role is an interesting problem that might contribute further in reducing the set of fundamental results in distributed computing.

Acknowledgements. The authors would like to thank Hagit Attiya and Keren Censor-Hillel for useful discussions and feedback on earlier versions of this paper.

References

[1] Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, STOC '95, pages 538–547, New York, NY, USA, 1995. ACM.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. An O(n log n) sorting network. In *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 1–9, New York, NY, USA, 1983. ACM.

[3] Dan Alistarh, James Aspnes, Keren Censor-Hillel, Seth Gilbert, and Morteza Zadimoghaddam. Optimal-time adaptive strong renaming, with applications to counting. In *PODC*, 2011.

[4] Dan Alistarh, Hagit Attiya, Seth Gilbert, Andrei Giurgiu, and Rachid Guerraoui. Fast randomized test-and-set and renaming. In *DISC*, pages 94–108, 2010.

[5] James Aspnes, Hagit Attiya, and Keren Censor. Max registers, counters, and monotone circuits. In *PODC*, pages 36–45, 2009.

[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] H. Attiya and D. Hendler. Time and space lower bounds for implementations using k-cas. *Parallel and Distributed Systems, IEEE Transactions on*, 21(2):162 –173, 2010.

[8] Hagit Attiya and Arie Fouren. Adaptive wait-free algorithms for lattice agreement and renaming (extended abstract). In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 277–286, New York, NY, USA, 1998. ACM.

[9] Hagit Attiya, Danny Hendler, and Philipp Woelfel. Tight RMR lower bounds for mutual exclusion and other problems. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, STOC '08, pages 217–226, New York, NY, USA, 2008. ACM.

[10] Michael Bender and Seth Gilbert. Mutual exclusion with O(log log n) amortized work. Under submission, 2011.

[11] Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *PODC*, pages 41–51, New York, NY, USA, 1993. ACM Press.

[12] Alex Brodsky, Faith Ellen, and Philipp Woelfel. Fully-adaptive algorithms for long-lived renaming. In *DISC*, pages 413–427, 2006.

[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] 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.

[15] Faith Ellen Fich, Danny Hendler, and Nir Shavit. Linear lower bounds on real-world implementations of concurrent objects. In *FOCS*, pages 165–173, 2005.

[16] Eli Gafni. The extended BG-simulation and the characterization of t-resiliency. In *STOC*, pages 85–92, 2009.

[17] Eli Gafni, Achour Mostéfaoui, Michel Raynal, and Corentin Travers. From adaptive renaming to set agreement. *Theor. Comput. Sci.*, 410:1328–1335, March 2009.

[18] Wojciech M. Golab, Vassos Hadzilacos, Danny Hendler, and Philipp Woelfel. Constant-RMR implementations of CAS and other synchronization primitives using read and write operations. In *PODC*, pages 3–12, 2007.

[19] Wojciech M. Golab, Danny Hendler, and Philipp Woelfel. An O(1) RMRs Leader Election Algorithm. *SIAM J. Comput.*, 39(7):2726–2760, 2010.

[20] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13:124–149, January 1991.

[21] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(2):858–923, 1999.

[22] Prasad Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '98, pages 201–210, New York, NY, USA, 1998. ACM.

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

[24] Yong-Jik Kim and James H. Anderson. A time complexity bound for adaptive mutual exclusion. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 1–15, London, UK, UK, 2001. Springer-Verlag.

[25] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[26] Shay Kutten, Rafail Ostrovsky, and Boaz Patt-Shamir. The Las-Vegas Processor Identity Problem (How and When to Be Unique). *J. Algorithms*, 37(2):468–494, 2000.

[27] Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25:1–39, 1995.

[28] 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.

[29] 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.

[30] Jae-Heon Yang and James Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9:51–60, 1995.

A Proofs Omitted from Section 3

Theorem 1. *For any $k \geq 1$, any wait-free deterministic adaptive renaming algorithm which renames into a namespace of size at most $2^{f(k)}$ for any function $f(k) = o(k)$ has a worst-case execution $\mathcal{E}$ in which (1) some process performs $\Omega(k)$ RMRs (and $\Omega(k)$ steps) and (2) each participating process performs a single* rename *operation.*

Proof. We begin by assuming for the sake of contradiction that there exists a deterministic adaptive algorithm R that renames into a namespace of size $M(k) = 2^{f(k)}$ for $f(k) \in o(k)$, with step complexity $C(k) = o(k)$. The first step in the proof is to show that any such algorithm can be transformed into a wait-free algorithm that solves adaptive *strong* renaming in the same model; the complexity cost of the resulting algorithm will be $O(C(k) + \log M(k))$.

Claim 1. *Any wait-free algorithm R that renames into a namespace of size $M(k)$ with complexity $C(k)$ can be transformed into a tight adaptive renaming algorithm $T(R)$ with complexity $O(C(k) + \log M(k))$.*

Proof. The first step in the proof is to consider a recursive AKS renaming network A, as defined in [3]. In brief, this data structure is built in two stages: the first stage is a recursive construction of a sorting network with an arbitrary number of inputs, which ensures the properties of a sorting network whenever truncated to any number of input (and output) ports; the second stage replaces each comparator in the sorting network with a (two-process) test-and-set object.

The key idea in the lower bound is that we can use the algorithm R to assign unique input ports to processes in the adaptive recursive AKS renaming network. More precisely, a process first calls the algorithm R to obtain a temporary name; it then uses this name as an input port for the adaptive AKS renaming network. Once it has an input port, the process starts at that port and follows a path through the network determined by leaving each comparator on its lower output wire, if it wins the test-and-set (i.e., returns 0 from the test-and-set invocation), and on the higher output wire otherwise (i.e., if it returns 1 from the test-and-set invocation); the process returns the index of its output port as its final name. When starting from an AKS sorting network, the recursive renaming network construction has the following properties, which can be shown starting from [3]: (1) if each process starts at a distinct input port, every correct process eventually reaches a *distinct* output port; (2) in every execution where k processes are assigned *distinct* input ports to the renaming network, the participants leave on the first k output ports of the network; (3) if I is the largest index of an occupied input port and O is the largest index of an occupied output port, then the step complexity of the resulting renaming algorithm is $O(\log \max(I, O))$.

Since algorithm R assigns unique names to processes in a namespace of size $M(k)$, from properties (1) and (2) we obtain that the composition of algorithm R and the AKS renaming network A solves adaptive strong renaming. Second, notice that $M(k)$, the size of the namespace generated by R is an upper bound on I, the largest index of an occupied input port in the renaming network A. Since R generates distinct names, and A solves adaptive strong renaming, we also have that $I \geq O$, the largest index of an occupied output port. We can now use property (2) to obtain that the step complexity of the renaming network algorithm is $O(\log \max(I, O)) = O(\log M(k))$. Thus, the total complexity of the algorithm is $O(C(k) + \log M(k))$. □

Returning to the main proof, in the case of algorithm R, the claim guarantees that the algorithm $T(R)$ solves strong adaptive renaming with complexity $o(k) + O(\log 2^{f(k)}) = o(k) + O(f(k)) = o(k)$.

The second step in the proof shows that any wait-free strong adaptive renaming algorithm can be used to solve adaptive mutual exclusion with only a constant blowup in terms of step or RMR complexity.

Claim 2. *Any deterministic algorithm R for adaptive strong renaming implies a correct adaptive mutual exclusion algorithm $ME(R)$. The RMR complexity of $ME(R)$ is upper bounded asymptotically by the RMR complexity of R, which is in turn upper bounded by its step complexity.*

Proof. First, note that the models in which renaming and mutual exclusion are generally considered are distinct: renaming assumes a wait-free read-write shared-memory model WF which may be augmented with atomic compare-and-swap or test-and-set operations; complexity is measured in the number of steps that a process takes during the execution. Mutual exclusion assumes a more specific cache-coherent (CC) or distributed shared memory (DSM) shared-memory model with no process failures (otherwise, a process crashing in the critical section could block the processes in the entry section forever); the complexity of mutex algorithms is measured in terms of remote memory references (RMRs). We call this second model the *failure-free, local spinning* model, in short LS.

The transformation from adaptive tight renaming algorithm R in the WF model to algorithm $ME(R)$ in the LS model proceeds as follows.

Description. The processes share an adaptive strong renaming object R, and a vector $Done$ of boolean bits, which are initially set to false. The vector $Done$ is of size n if n is known, otherwise its size is unbounded.

First, we consider the subset of registers used by the algorithm R on which either (atomic) compare-and-swap or test-and-set operations may be performed during an execution. We replace every such register with a read-write compare-and-swap implementation with constant RMR complexity using the techniques presented in [18, 19]. This compare-and-swap implementation also supports atomic read and write operations. We have thus obtained a read-write adaptive strong renaming algorithm R' in the LS model whose RMR complexity is the same (modulo constant factors) as the step complexity of R.

After this transformation, the processes use the algorithm R' and the vector of boolean bits to solve mutex adaptively. (We recall that the *adaptive* version of the mutex problem requires the complexity to depend on k, not on n.) For the enter procedure of the mutex implementation, each of the k participating processes runs the algorithm R', and obtains a unique name from 1 to k. The process that obtained name 1 enters the critical section; upon leaving, it sets the $Done[1]$ bit to true. Any process that obtains a name $id \geq 2$ from the adaptive renaming object spins on the $Done[id-1]$ bit associated to name $id-1$, until the bit is set to true. When this occurs, the process enters the critical section. When calling the exit procedure to release the critical section, each process sets the $Done[id]$ bit associated with its name to true and returns.

Analysis. We now show that the above transformation generates correct mutual exclusion algorithms whose RMR complexity is upper bounded the step complexity of the adaptive renaming algorithm.

We begin by proving the correctness of the resulting mutual exclusion protocol. First, notice that the algorithm R' obtained from the first stage of the transformation is a correct adaptive strong renaming algorithm, by the correctness of the algorithm R and of the read-write compare-and-swap implementations in [18, 19].

Second, since the algorithm R is *wait-free*, and the resulting algorithm R' executes in a model where no process may crash, the algorithm R' has the property that, under a fair scheduler that continues to schedule each process until it returns, eventually, every process returns a name from the algorithm R'.

To check the *mutual exclusion* property, first let k be the number of participating processes in the current execution. Since the algorithm R' is a correct adaptive strong renaming algorithm, every participating process eventually obtains a distinct name from 1 to k. Notice now that, by the structure of the protocol, the process that obtained name $v+1$ from R' may only enter the critical section *after* the process with name v has performed the $Done[v] = \mathsf{true}$ operation in its exit procedure. Hence, the two processes may not be in the critical section at the same time. Since no two processes may obtain the same name, the mutual exclusion property follows from a simple induction argument.

For *deadlock freedom*, notice that, since every process is eventually scheduled until it returns, then every process obtains a name from algorithm R'. Since the namespace is tight and adaptive, for each name v between 1 and k, there exists a process p_v that eventually obtains that name. In particular, there exists a process that obtains name 1. This process eventually enters the critical section, and then sets the $Done[1]$ bit to true. Consider now process p_2 that obtained name 2. Eventually, this process sees the $Done[1]$ bit as true, therefore it enters the critical section, then leaves it and sets the $Done[2]$ bit to true. We can continue the induction to obtain that, eventually, every process p_k enters the critical section. The *finite exit* property follows trivially.

For the complexity claims, first notice that, since we replaced every compare-and-swap call in R with an implementation requiring a constant number of RMRs, the RMR complexity of the algorithm R' is at most a constant times the RMR complexity of the algorithm R. (The transformation from R to R' is a special case of the generic transformation presented in [18].)

Next, notice that, once a process obtains the name from algorithm R', it performs exactly one extra RMR before entering the critical section, since an RMR is charged only when it sees the true value in the $Done[v-1]$ register, which may occur at most once. Therefore, the (local or global) RMR complexity of the mutex algorithm is at most a constant times the (local or global) RMR complexity of the original algorithm R. Since the algorithm R is wait-free, therefore uses no busy-waiting loops, its RMR complexity is a lower bound on its step complexity, which concludes the proof of the claim. □

To conclude the proof of Theorem 1, we notice that the algorithm that results from the composition of the two claims, that is, $M(T(R))$, is an adaptive mutual exclusion algorithm with RMR complexity $o(k) + O(f(k)) = o(k)$. However, the existence of this algorithm contradicts the $\Omega(k)$ lower bound on the RMR complexity of adaptive

mutual exclusion by Anderson and Kim [24]. The contradiction arises from our initial assumption on the existence of algorithm R. The claim about the number of rename operations follows from the structure of the transformation and from that of the mutual exclusion lower bound of [24]. The claim about the number of rename operations follows from the structure of the transformation and from that of the mutual exclusion lower bound of [24].

$\square_{Theorem\ 1}$

Corollary 3. *For any $n \geq 1$, any wait-free deterministic (non-adaptive) renaming algorithm that renames into a polynomial namespace has an execution in which a process performs $\Omega(n)$ steps.*

Proof. The proof is based on a new reduction between non-adaptive and adaptive renaming that preserves polynomial namespace size and asymptotic step complexity; we then apply Theorem 1 to obtain the linear lower bound.

Assume for the sake of contradiction that there exists a non-adaptive renaming algorithm R such that, for any n, there exists $c \geq 1$ such that the algorithm R accepts n participants to which it assigns names between 1 and n^c in any execution, and R has worst-case step complexity $o(n)$. (The size of the namespace does not depend on the number of participants, as the algorithm is non-adaptive.)

Description. We start from this non-adaptive algorithm and obtain an adaptive renaming algorithm with similar namespace and complexity bounds. The transformation is described in Algorithm 2. The processes share a counter with poly-logarithmic complexity as introduced in [5], and a series of renaming algorithms $(R_i)_{i\geq 1}$, such that R_i is an instance of algorithm R to which only 2^i processes may participate. More precisely, for every $i \geq 1$, we obtain R_i by instantiating R for $n = 2^i$. It follows that algorithm R_i will assign names between 1 and 2^{ci}.

A process first increments the counter C, then reads its current value v. It then computes the index of the renaming object it is going to access to be $\lceil \log v \rceil$. The process then accesses the renaming object $R_{\lceil \log v \rceil}$, from which it obtains a name. The process returns the sum between this name and the namespace size generated by previous R_i instances, that is $\sum_{i=1}^{index-1} 2^{ci}$.

Analysis. We begin by showing that the transformation is a correct renaming algorithm. First, we show that none of the renaming instances R_i gets overwhelmed, i.e. at most 2^i processes access instance R_i, for any i. This follows since at most 2^i participating processes can obtain values $v \leq 2^i$ from the shared counter, as the counter is exact and linearizable. Second, the transformation returns unique names, since the namespace is partitioned into namespaces returned by the instances R_i. The transformation terminates since all its components are wait-free for a finite number of participants.

We now bound the size of the namespace that the algorithm generates as a function of k, the number of participants. Let M be the largest $index$ that a participant obtains in line 7. Then the size of the namespace is bounded by $\sum_{i=1}^{M} 2^{ci} \leq 2^{c(M+1)}$. On the other hand, since the counter is linearizable and exact, $M \leq \lceil \log k \rceil$. We then obtain that the size of the namespace generated by the transformation is at most $(2k)^c$, i.e. polynomial in k. Therefore, the transformation is an adaptive renaming algorithm that renames into a namespace polynomial in k.

Finally, we bound the step complexity of the construction. Consider a process p; without loss of generality, we can assume that p gets a name in the largest, most expensive, renaming instance that processes access in this execution, that is $R_{\lceil \log k \rceil}$. Then the number of steps that p performs is $O(\log^2 k + C(R_{\lceil \log k \rceil}))$, where the second factor is the complexity of $R_{\lceil \log k \rceil}$. Since $R_{\lceil \log k \rceil}$ accepts $\Theta(k)$ participants, by assumption its step complexity is $o(k)$.

Then the step complexity of the resulting adaptive renaming algorithm is $O(\log^2 k) + o(k) = o(k)$. Since the algorithm assigns names in a namespace polynomial in k, we obtain a contradiction with Theorem 1, from which the claim follows. (We note that the parameter n in the renaming algorithm is independent from the parameter n used in the mutual exclusion lower bound.)

$\square_{Corollary\ 1}$

B Proofs Omitted from Sections 4 and 5

Lemma 3. *Let $\mathcal{E}$ be an execution in $\mathcal{S}(A)$ and p be a process with $UP(p, r) = S$ at the end of round r. There exists an execution $\mathcal{E}'$ of A in which only processes in S take steps, such that $\mathcal{E}$ and $\mathcal{E}'$ are* indistinguishable *to process p.*

Proof. To prove the claim, we provide an algorithm for the adversary to build an execution $\mathcal{E}'$ based on the original execution $\mathcal{E} \in \mathcal{S}(A)$, and prove that $\mathcal{E}'$ and $\mathcal{E}$ are indistinguishable for process p at the end of round r. We note that

```
Parameters: a series of non-adaptive renaming algorithms (R_i)_{i≥1},
A shared counter C with poly-logarithmic complexity [5]
For any i, the algorithm R_i accepts at most 2^i participating processes and assigns a namespace of size 2^{ci}
procedure adaptive-rename()
    C.increment()
    v ← C.read()
    index ← ⌈log v⌉
    return Σ_{i=1}^{index−1} 2^{ci} + R_index.rename()
```

Algorithm 2: The transformation from non-adaptive to adaptive algorithms.

```
Parameters: the execution E, the set S
procedure build(E, S)
    r ← 1
    while true do
        // we schedule processes that have not seen a process outside of S
           in the first r − 1 rounds of E
        S_r ← { processes q | UP(q, r − 1) ⊆ S }
        for each process q in S_r do
            process q performs coin tosses until it returns or has enabled a shared-memory operation
            the jth coin toss by process q is supplied with outcome coins(E, q, j)
        R ← processes in S_r that have read operations enabled
        W ← processes in S_r that have write operations enabled
        C ← processes that have compare-and-swap operations enabled
        schedule all processes in R to perform their operations, in the order of their initial identifiers
        schedule all processes in W to perform their operations, in the order of their initial identifiers
        schedule all processes in C to perform their operations
        in the order defined by the secretive schedule σ
        r ← r + 1
```

Algorithm 3: The procedure for building the indistinguishable execution.

the construction is an adaptation to the read-write model of the one presented by Jayanti in [22]. Given the execution $\mathcal{E}$, let $\mathsf{coins}(\mathcal{E}, p, j)$ be the outcome of the jth coin toss that process p performs in execution $\mathcal{E}$.

Constructing the execution. The procedure to build the desired execution $\mathcal{E}'$ of algorithm A in which only $|S|$ processes participate is described in Algorithm 3. The run is also structured in rounds. Of note, only processes that are scheduled in round r are the processes in S that have not witnessed processes outside of S by the end of round $r-1$. Each process is scheduled to perform local coin tosses until it has a shared-memory operation enabled. For every coin toss j by a process q, the adversary feeds the outcome that occurred in the execution $\mathcal{E}$, that is $\mathsf{coins}(\mathcal{E}, q, j)$. Depending on their enabled shared-memory operation, the processes that have not yet terminated are then split into a set of readers and a set of writers. The readers are then scheduled in the order of their initial identifiers, after which the writers are scheduled in the order of their initial identifiers. Finally, the adversary increments the round counter and moves to the next round.

Correctness of the construction. The proof of correctness proceeds by induction on the round number r, and is similar to the one outlined in [22], Lemma 5.2, in the case of the LL/SC model, where the execution $\mathcal{E}$ is the $(All, \mathcal{A})$ run, and the execution $\mathcal{E}'$ is the $(S, \mathcal{A})$-run. We therefore refer the reader to reference [22] for the proof.

$\square_{Lemma\ 3}$

Proposition 1 (Expected Complexity)**.** *Fix constants $\alpha \in [0, 1]$ and $\gamma > 0$. Given an algorithm A that terminates with probability α, if there exists an adversarial strategy $\mathcal{S}(A)$ such that, in every execution under $\mathcal{S}(A)$ in which every*

process terminates, the processes take at least γ steps, then the (worst-case) expected step complexity of A is at least $\alpha\gamma$.

Theorem 3. *Fix $c \geq 1$ constant. Let A be a linearizable c-approximate counter implementation that terminates with probability α. For any k, the algorithm A has worst-case expected total step complexity $\Omega(\alpha k \log(k/c^2))$, in runs where each process performs an* increment *followed by a* read *operation.*

Proof. Let A be a c-approximate counting algorithm in this model. We consider *terminating* executions $\mathcal{E}$ with k participating processes, in which each process performs an increment operation followed by a read operation, during which which the adversary applies the strategy described in Algorithm 1, i.e. $\mathcal{E} \in \mathcal{S}(A)$.

Again, we start by noticing that, since no process crashes during $\mathcal{E}$, each process has to return a value from the read operation. Depending on the linearization order of the increment and read operations, the processes may return various values from the read. Let γ_i be the number of increment operations linearized *before* the read operation by process p_i, and let v_i be the value returned by process p_i's read. Without loss of generality, we will assume that the processes p_i and their return values v_i are sorted in the increasing order of their γ_i values.

First, notice that, since every process calls increment before its read operation, for every $1 \leq i \leq k$, $\gamma_i \geq i$. Second, by the c-approximation property of the counter implementation, $v_i \geq \gamma_i/c$. Therefore, $v_i \geq i/c$.

Second, consider process p_i returning value v_i in $\mathcal{E}$. Let ℓ_i be the number of shared-memory operations that p_i has performed in $\mathcal{E}$. Since the adversary schedules each process once in every round of $\mathcal{E}$, it follows that process p_i has returned at the end of round ℓ_i. Let $S = UP(p_i, \ell_i)$. By Lemma 2, we have that $|S| \leq 4^{\ell_i}$.

Assume for the sake of contradiction that $|S| < v_i/c$. By Lemma 3, there exists an execution $\mathcal{E}'$ of A which is indistinguishable from $\mathcal{E}$ from q_i's point of view at the end of round ℓ_i, in which only $|S| < v_i/c$ processes take steps. However, since the counter is c-approximate, the highest value that process p_i can return in execution $\mathcal{E}'$, and thus in $\mathcal{E}$, is *strictly less* than $c \cdot (v_i/c) = v_i$, a contradiction.

Therefore, $|S| \geq v_i/c$, and therefore $\ell_i \geq \frac{1}{2}\log(v_i/c) \geq \frac{1}{2}\log(i/c^2)$, for every $1 \leq i \leq k$. We obtain that

$$\frac{1}{2}\sum_{i=1}^{k} \ell_i \geq \frac{1}{2}\sum_{i=1}^{k} \log(i/c^2) \geq \frac{1}{2}\log(k!/c^{2k}) = \Omega(k \log(k/c^2)).$$

Using Proposition 1, we obtain the claim. $\square_{Theorem\ 3}$

Lemma 4 (Reductions)**.** *Given either a linearizable fetch-and-increment register, or a queue or stack that can hold n elements, one can solve adaptive strong renaming using one operation per process.*

Proof. Given a linearizable fetch-and-increment register, we can trivially solve adaptive strong renaming by having each participant call the fetch-and-increment operation once, and return the value received plus 1.

Given a linearizable shared queue initialized with n distinct objects $1, 2, \ldots, n$, we can solve adaptive strong renaming by having each participant call the dequeue operation once, and return the value received. The transformation is similar for a stack initialized with elements $1, 2, \ldots, n$. $\square$