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 an *individual* 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 individual 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* [7], 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. [7, 12, 13, 29, 30] or using randomization [3, 5, 16, 31]. Herlihy and Shavit [23], as well as Rajsbaum and Castañeda [14], showed that, for deterministic algorithms using only reads and writes, 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, e.g. [3, 29], are able to circumvent this impossibility and assign a *tight* namespace of k consecutive names to the k participants.

Little is known about renaming lower bounds. Establishing such bounds 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 in a namespace whose size depends on the number of participants (this is also called *adaptive* renaming [8]). Our study covers both randomized and deterministic algorithms, and also implies lower bounds for other shared objects such as counters, stacks, and queues.

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. 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 shows that assigning names 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 bound is tight for algorithms that use atomic compare-and-swap operations [29]. For read-write algorithms, which cannot achieve a tight namespace, the bound is matched by the algorithm of [30], which ensures a namespace of size $O(k^2)$. The bound highlights an exponential complexity separation between deterministic and randomized adaptive renaming, since there exist randomized renaming algorithms

Shared Object	Lower Bound	Type	Matching Algorithms	New Result
Deterministic c-loose Renaming	$\Omega(k)$	Local	[8,29,30]	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	[6]	Yes
Fetch-and-Increment	$\Omega(k)$	Local	[3,29]	Improves on [18]
	$\Omega(k \log k)$	Global	[3]	Improves on [9]
Queues and Stacks	$\Omega(k)$	Local	Universal Constructions [1,22]	Improves on [18]
	$\Omega(k \log k)$	Global	-	Improves on [9]

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

with $O(\log k)$ expected local (per-process) step complexity [3].

The key idea behind the lower bound is a new reduction between renaming and mutual exclusion. The strategy is composed of two steps (please see Figure 2 for a description). The first reduction step transforms any wait-free adaptive loose renaming algorithm R into an algorithm for wait-free *strong* adaptive renaming. The transformation preserves the asymptotic step complexity of algorithm R as long as R renames into a namespace of size sub-exponential in the number of participants k. The second step of the reduction uses the resulting strong renaming algorithm to solve mutual exclusion, with only a constant blowup in terms of complexity. Overall, the transformation ensures that, if the algorithm R renames in a sub-exponential namespace with step complexity $o(k)$, the resulting mutual exclusion algorithm uses $o(k)$ *remote memory references* (RMRs) [10, 26] to enter and exit the critical section. However, the existence of such an algorithm contradicts a linear lower bound of Anderson and Kim [26] on the RMR complexity of mutual exclusion, concluding the argument. One side result of the reduction is a new mutual exclusion algorithm, based on an AKS sorting network, which is asymptotically time-optimal.

More generally, our reduction connects the complexity of wait-free algorithms that use reads, writes, and synchronization primitives such as compare-and-swap or test-and-set, with the complexity of locking implementations that rely on busy-waiting loops. In fact, our reduction technique implies a stronger $\Omega(k)$ lower bound on the number of RMRs that a process has to perform in worst-case executions. RMRs are orders of magnitude slower than accesses to local memory on most multi-processor architectures. Since renaming can be achieved deterministically wait-free with $O(k)$ RMRs [30], this also 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 [18, 25]. 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 compare-and-swap is 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 [18, 25].

We complement our individual 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].

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

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

technique improves on previous lower bounds for these objects [9, 10, 24], since it covers the *global* complexity of *randomized approximate* solutions. Our global results are tight within logarithmic factors for counters [6] and fetch-and-increment registers [3]. In the case of counters, since the lower bounds apply to randomized algorithms, and the almost-matching algorithm [6] is 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; complete proofs can be found in the full version of this paper [4].

2 Preliminaries

Assumptions. We consider a standard asynchronous shared memory model with n processes, $t < n$ of which may fail by crashing. Processes that do not crash during the execution are called *correct*. In the following, we assume that n is unbounded. Algorithms that work in this model are called *wait-free*. 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 atomic registers. Each register R exports atomic read, write, and compare-and-swap operations, with the usual semantics, and allows multiple readers and multiple writers. 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 [7] 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 $c \geq 1$.

Alternatively, renaming can be seen as an object with unboundedly many input and output ports. Each input port is associated to an initial identifier, and the object eventually assigns a unique output port to each correct process. The renaming object is *adaptive* if it assumes no bound on the number of processes that may participate in an execution, and if the size of the range of assigned output ports (i.e., the target namespace) depends only on the contention k. (By contrast, a *non-adaptive* renaming object assumes a known parameter n that bounds the number of participating processes (and thus also the number of input ports that may be occupied in an execution). Also, the range of assigned output ports is a function of the parameter n.)

The *counter* object exports operations increment and read. The increment operation increments the value of the counter by one, and returns a *success* indication. The read operation returns the number of increment operations that precede it in the linearization order. A c-approximate counter (also called a c-multiplicative-approximate counter) 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 that precede the read operation in the linearization order.

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 that invokes the enter operation will complete 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.

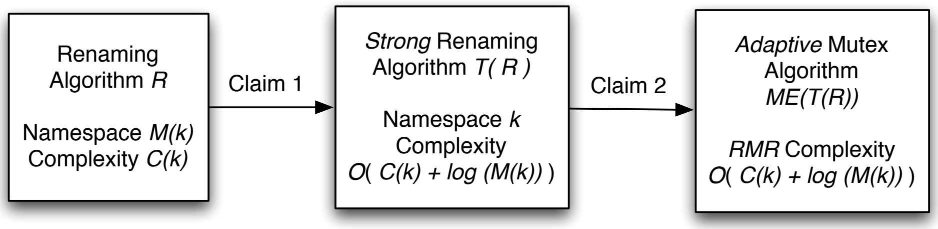
Renaming Algorithm R
Namespace M(k) Complexity C(k)
Claim 1
Strong Renaming Algorithm T(R)
Namespace k Complexity O(C(k) + log (M(k)))
Claim 2
Adaptive Mutex Algorithm ME(T(R))
RMR Complexity O(C(k) + log (M(k)))

Figure 2: The structure of the reduction in Theorem 1. When $C(k)$, the complexity of algorithm R, is in $o(k)$, and $M(k)$, the target namespace for R, is a sub-exponential function, the RMR complexity of the resulting adaptive mutex algorithm $ME(T(R))$ is $o(k)$, contradicting a lower bound by Anderson and Kim [26].

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 *individual 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) [10, 26]. 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 (or no copy of the variable); 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 [10, 26]. For wait-free algorithms, such as the renaming algorithms we consider, the RMR complexity is always a lower bound on their step complexity.

3 The Main Lower Bound

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. The technique establishes a reduction between renaming and mutual exclusion using optimal-depth AKS sorting networks [2] as an intermediate step.

The Strategy. The proof is based on two steps, outlined in Figure 2. The first step, contained in Claim 1, starts from a wait-free algorithm R, renaming adaptively into a loose namespace of sub-exponential size $M(k)$, and obtains an algorithm $T(R)$ for *strong* adaptive renaming. The extra complexity cost of this step is an additive factor of $O(\log M(k))$. The second step, contained in Claim 2, uses the strong renaming algorithm $T(R)$ to solve adaptive mutual exclusion, with the property that the RMR complexity of the resulting adaptive mutual exclusion algorithm $ME(T(R))$ is $O(C(k) + \log M(k))$, where $C(k)$ is the step complexity of the initial algorithm R. Finally, we revert to an $\Omega(k)$ lower bound on the RMR complexity of adaptive mutual exclusion by Anderson and Kim [26]. When plugging in any sub-exponential function for $M(k)$ in the expression bounding the RMR complexity of the adaptive mutual exclusion algorithm $ME(T(R))$, we obtain that the algorithm R must have step complexity at least linear in k, which concludes the proof.

We make this argument precise in the following. Due to space limitations, we only present proof sketches for some of the claims, and defer the complete proofs to the full version of this paper [4].

Theorem 1 (Individual Lower Bound). *Given n unbounded, for any $k \geq 1$, any wait-free deterministic adaptive renaming algorithm that renames into a namespace of size at most $2^{f(k)}$ for any function $f(k) = o(k)$ has a worst-case execution with $2k - 1$ participants in which (1) some process performs k RMRs (and 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. We start from a recursive data structure based on AKS sorting networks [2], called an *adaptive renaming network* [3]. An adaptive renaming network is a construction with an unbounded number of input and output 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. The structure and properties of renaming networks have been introduced in [3]. We now present a reduction from loose renaming to strong renaming based on renaming networks.

Description. Given an AKS adaptive renaming network A, we first 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 higher output wire, if it wins the test-and-set (i.e. returns 0 from the test-and-set invocation), and on the lower output wire otherwise (lower wires have higher index, as wire indices are assigned from top to bottom). The process returns the index of its output port as its final name in the algorithm $T(R)$.

Analysis. We recall that renaming networks have the property that, for any k, they solve strong adaptive renaming for k processes [3, Theorem 3]. Also, if the range of occupied input ports is bounded by a parameter $M > 0$, then a renaming network has depth $O(\log M)$ [3, Corollary 3].

Based on these properties, it follows 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 $O(\log \max(M(k), O))$. (For a complete derivation, please see the full version of this paper [4].)

Properties (1) and (2) ensure that the composition of R and A solves strong adaptive renaming. Properties (2) and (3) imply that the number of steps a process takes while executing 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 assumed algorithm R, the claim guarantees that the resulting 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 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. We begin by noting a few key distinctions between renaming and mutual exclusion. Renaming algorithms are usually wait-free, and assume a read-write shared-memory model 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. For simplicity, in the following, we call this the *wait-free* (WF) model. 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 would block the processes in the entry section forever). Thus, solutions to mutual exclusion are inherently blocking; 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 uses the algorithm R to solve mutual exclusion. The key idea is to use the names obtained by the processes as tickets to enter the critical section.

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. Since the algorithm R is wait-free, it can be run in the LS model with no modifications.

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. (This construction is designed for the CC model; a similar construction exists for the DSM model.)

The fact that algorithm $ME(R)$ is a correct mutex implementation follows from the tightness and adaptivity of the namespace generated by algorithm R. 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 at most two extra RMRs before entering the critical section, since RMRs may be charged only when first reading the $Done[v-1]$ register, and when the value of this register is set to true. Therefore, the (individual or global) RMR complexity of the mutex algorithm is the same (modulo constant additive factors) as the RMR complexity of the original algorithm R. Since the algorithm R is wait-free, 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 that requires $o(k) + O(f(k)) = o(k)$ RMRs to enter and exit the critical section. However, the existence of this algorithm contradicts the $\Omega(k)$ lower bound on the RMR complexity of adaptive mutual exclusion by Anderson and Kim [26, Theorem 2]. More precisely, they showed that, given $k > 0$, for $n = \Omega(k^{2^k})$, any *deterministic* mutual exclusion algorithm using reads, writes, and compare-and-swap operations, that accepts at least n participating processes has a computation involving $(2k-1)$ participants in which some process performs k remote memory references to enter and exit the critical section [26].

Since the algorithm R is adaptive and therefore works for unbounded n, the adaptive mutual exclusion algorithm $ME(T(R))$ also works for unbounded n. Hence, the above mutual exclusion lower bound contradicts the existence of algorithm $ME(T(R))$. 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 [26]. □

Relation between k and n. The lower bound of Anderson and Kim [26] 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. (For example, the lower bound does not preclude an algorithm with running time $O(\min(k, \log n))$ if n is known in advance.) On the other hand, the lower bound applies to all algorithms that work for arbitrary values of n.

Read-write algorithms. Notice that, although the first reduction step employs compare-and-swap (or test-and-set) operations for building the renaming network, the lower bound also holds for algorithms that only employ read or write operations.

Progress conditions. Many known adaptive renaming algorithms, e.g. [3, 30], do not guarantee wait-freedom in executions where the number of participants is unbounded, since a process may be prevented from acquiring a name indefinitely by new incoming processes. Note that our lower bound applies to these algorithms as well, as the original mutual exclusion lower bound of Anderson and Kim [26] applies to all mutex algorithms ensuring livelock-freedom, and our transformation does not require a strengthening of this progress condition.

3.1 Non-Adaptive Renaming Lower Bound

This technique also implies an almost-linear lower bound on the step complexity of *non-adaptive* renaming algorithms. Recall that, for non-adaptive algorithms, the size of the set of participants is bounded by a fixed parameter n, and the size of the resulting namespace depends on this parameter.

We define a renaming algorithm *with overflows* as a non-adaptive renaming algorithm RF, that has the same specification as a renaming algorithm as long as the number of participants k does not exceed the maximum number of participants n. On the other hand, if $k > n$, then the algorithm RF may return a special value overflow to the calling process instead of a (unique) name. We call an *instance* of the algorithm a variant of RF for n fixed. Note that an instance assumes no limit on the size of the initial identifiers that participanting processes may have; however, a non-overflow return value is guaranteed only if at most n processes participate.

From non-adaptive renaming to renaming with overflows. In the following, we consider non-adaptive renaming algorithms that ensure the following three properties: (1) every step of the algorithm executed by a correct process terminates eventually; (2) any step can only update variables that are part of the algorithm; (3) the algorithm works for unbounded namespace size. It is easy to see that any wait-free algorithm can be modified to ensure properties (1) and (2), while property (3) is given by the original specification of the (non-adaptive) renaming problem, as given in e.g. [7].

Given these properties, one can transform any non-adaptive renaming algorithm R that accepts an initial input namespace of unbounded size into a renaming algorithm with overflows RF with the same asymptotic complexity by the following procedure. Let $M(n)$ be the namespace that R renames into, and let $C(n)$ be R's complexity. We associate with each process a local program counter that counts the number of steps that the process has performed in the current execution. If the counter exceeds the value $C(n)$ while running R, then the process automatically returns overflow. If the process returns a name that is larger than $M(n)$ from R, then it automatically returns overflow. Finally, if the process obtains a name r from the current instance of R, then it checks that this name is unique by accessing an auxiliary array of *splitter* objects, as presented in e.g. [30], in position r. If the splitter returns stop, then the process returns that name as its decision value (the splitter properties [30] ensure that only one process may return stop at the same splitter). Otherwise, the process returns overflow.

It is easy to check that this transformation results in a renaming algorithm with overflows, whose asymptotic step complexity is the same as the one of the original algorithm R. Although, in general, the behavior of a non-adaptive renaming algorithm is not specified when k exceeds n, this transformation is natural, and works for all known non-adaptive renaming algorithms that do not assume an upper bound on the size of the initial namespace.

Therefore, in the following, we consider renaming algorithms with overflows. We show that any renaming algorithm with overflows can be transformed into an *adaptive* renaming algorithm, at the cost of a multiplicative logarithmic increase in running time, conserving polynomial namespace size. In turn, based on our previous lower bound, this will imply a lower bound of $\Omega(n)$ on the running time of any deterministic renaming algorithm with overflows. We now present the transformation from renaming with overflows to adaptive renaming.

From renaming with overflows to adaptive renaming. We start from a non-adaptive renaming with overflows algorithm R, which, for any $n \geq 1$, renames into a namespace of size $M(n)$, with complexity $C(n)$, as long as the number of participants k to the instance does not exceed n. We consider an infinite series $(R_i)_{i=1,2,\ldots}$ of instances of algorithm R, where instance R_i is the algorithm R for parameter $n = 2^i$. The transformation proceeds as follows. Each process accesses the instances $(R_i)_i$ in order, until it first obtains a name from an instance R_i (as opposed to an overflow indication). If, on the other hand, a process obtains an overflow indication from R_i, it increments its instance counter i, and proceeds to the next instance. Once it has obtained a name v, the process returns v plus the sum resulting from adding up the namespace sizes for the previous instances, i.e., for $j \geq 2$, $\sum_{j=1}^{i-1} M(2^j)$.

We now prove that the algorithm described above is a correct adaptive renaming algorithm, and bound its complexity and namespace size.

Lemma 1. *Let A be an algorithm that renames with overflows such that, for any $n \geq 1$, it guarantees a namespace of size polynomial in n with step complexity $o(n)$. Then the above transformation yields an* adaptive *renaming algorithm that renames in a namespace polynomial in the number of participants k, whose complexity is $o(k)$.*

Proof. Fix an arbitrary execution of the transformation, and let k be the number of participants in the execution. Let m be highest index of an instance in the series $(R_i)_i$ that a process accesses in this execution. Since a process may only access an instance of a higher index if it overflows in the current instance, and an instance R_i may return overflow only if the number of participants k exceeds the number of allowed participants 2^i, it follows that $m = O(\log k)$.

It follows that, for bounded k, each correct process eventually returns a name in the transformation. (On the other hand, if k is infinite, then the transformation no longer guarantees starvation-freedom.) The name uniqueness property follows since renaming with overflows guarantees uniqueness, and the namespace resulting from the transformation is partitioned into the namespaces returned by the instances $(R_i)_i$.

We now bound the size of the namespace that the algorithm generates as a function of k, the number of participants. Assume that, for any n, the algorithm R returns names between 1 and n^c. If m is the largest index of an accessed instance, then the size of the namespace is bounded by $\sum_{i=1}^{m} 2^{ci} \leq 2^{c(m+1)} = O(k^c)$, i.e. polynomial in k. Also, notice that the transformation uses no knowledge of n. Therefore, the transformation is an *adaptive* renaming algorithm that renames into a namespace polynomial in the contention k.

Finally, we bound the step complexity of the transformation. By its structure, the number of steps a process takes in total is bounded by $C(2^m)+C(2^{m-1})+C(1)$. Since R_m is the highest accessed instance of algorithm R, it follows that $2^{m+1} > k \geq 2^m$. On the other hand, since we can assume naturally that the complexity function $C(\cdot)$ is monotonically increasing, we obtain that the step complexity of the transformation is bounded by $C(2k)+C(k)+C(k/2)+\ldots+C(1)$. Since $C(n) = o(n)$, we obtain by a change of variables that the step complexity of the transformation is $o(k)$, as claimed. □

On the other hand, the existence of such an adaptive renaming algorithm contradicts Theorem 1, given that we can run the resulting adaptive algorithm in a system with no upper bound on the number of participating processes. Therefore, it follows that every deterministic renaming algorithm with overflows has complexity $\Omega(n)$. Note that, due to our first transformation, the same result holds for non-adaptive renaming algorithms.

Corollary 1. *Any deterministic non-adaptive renaming algorithm, with the property that for any $n \geq 1$ the algorithm ensures a namespace polynomial in n, has worst-case step complexity $\Omega(n)$.*

3.2 A Time-Optimal Non-Adaptive Mutex Protocol

One 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 [20,21]. 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. [10]. Anderson and Yang [32] 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 [27], 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

```
procedure adversarial-scheduler()
$r \leftarrow 1$
while true do
    for each process $p$ do
        schedule $p$ to perform coin flips until it has enabled a shared-memory operation, or $p$ returns
    $\mathcal{R} \leftarrow$ processes that have read operations enabled
    $\mathcal{W} \leftarrow$ processes that have write operations enabled
    $\mathcal{C} \leftarrow$ processes that have compare-and-swap operations enabled
    schedule all processes in $\mathcal{R}$ to perform their operations, in the order of their initial identifiers
    schedule all processes in $\mathcal{W}$ to perform their operations, in the order of their initial identifiers
    schedule all processes in $\mathcal{C}$ to perform their in the order defined by the secretive schedule $\sigma$
    $r \leftarrow r + 1$
```

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

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 2, 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 [24]. We re-state the definition and properties of secretive schedules [24] in our model.

Lemma 2 (Secretive Schedules [24]). *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, by reading the register R at the end of round r. Our notation follows the one in [24], 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, as the rounds progress. The proof follows by induction on the round number $r \geq 0$.

Lemma 3 (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 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 [24] in the context of local lower bounds in the LL/SC model. Therefore, we omit this technical step in this extended abstract (an outline of the construction can be found in the full version of this paper [4]).

Lemma 4 (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 3, 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 4, 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.

Therefore, it has to hold 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 $\log_4(m_i/c) = \frac{1}{2}\log(m_i/c)$. 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 [6] 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.

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.*

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 3, we have that $|S| \leq 4^{\ell_i}$.

Assume for the sake of contradiction that $|S| < v_i/c$. By Lemma 4, 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 $\ell_i \geq \frac{1}{2}\log(v_i/c) \geq \frac{1}{2}\log(i/c^2)$, for every $1 \leq i \leq k$. We obtain that the total number of steps is bounded as follows.

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

□

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. This implies local and global lower bounds for these objects.

Corollary 2 (Applications)**.** *Given a system with unbounded n, 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 $2k - 1$ participants in which (1) each participant performs a single call, and (2) some process performs k RMRs (or steps).*
- *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 [7], 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 [23], with refinements by Rajsbaum and Castañeda [14]. This lower bound can be circumvented using hardware compare-and-swap or test-and-set operations [29], as well as using randomization [16] (at the cost of allowing a vanishing probability that the algorithm does not terminate). Renaming has been shown to be related to weak symmetry breaking in [19]; it is also related to the processor identity problem [28]; 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, 5, 7, 8, 12, 15, 29, 30]. The randomized renaming algorithm of [3] is time-optimal, as per our global lower bound. On the other hand, our local lower bound is matched by the algorithm of [29] which assumes hardware test-and-set operations; for read-write algorithms, it is matched by the algorithm of [30], which achieves a namespace of size $O(k^2)$; it is also matched (within a logarithmic factor) by the algorithm of [8], which ensures a linear namespace of size $(6k - 1)$. 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 [24]. Our lower bound generalizes Jayanti's result [24] in two ways: first, since we consider *total* step complexity, our results imply the local bounds of [24]; second, we consider the *loose* version of the problem, which relaxes the tight namespace requirements. Chlebus and Kowalski [15] also show a linear lower bound for deterministic renaming algorithms under the assumption that the number of available registers is limited. (By contrast, our lower bounds do not require such a restriction.)

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 [25], as well as Ellen et al. [18], 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 [6]. Our linear local bound does not have this limitation, since each process performs only one operation in the worst-case execution (note that, consequently, our deterministic linear lower bound does not apply to counters). 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 deterministic lower bounds of Attiya et al. [9, 10] and that of Fan and Lynch [17] for the state-change model, when applied to these objects.

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 [11] that approximate counting can be achieved with $O(\log \log n)$ expected steps against an oblivious adversary, that fixes the schedule in advance.

Our results reveal 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.

8 Acknowledgements

The authors would like to thank Hagit Attiya and Keren Censor-Hillel for discussions and feedback on earlier versions of this paper, and the anonymous reviewers for their useful suggestions.

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] Miklós Ajtai, János Komlós, and Endre Szemerédi. An O(n log n) Sorting Network. In *STOC*, pages 1–9. ACM, 1983.

[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, James Aspnes, Seth Gilbert, and Rachid Guerraoui. The Complexity of Renaming. Technical report, EPFL, 2011.

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

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

[7] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Ruediger Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, 1990.

[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 and Danny Hendler. Time and space lower bounds for implementations using k-CAS. *Parallel and Distributed Systems, IEEE Transactions on*, 21(2):162 –173, February 2010.

[10] 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.

[11] Michael Bender and Seth Gilbert. Mutual exclusion with O(log log n) amortized work. In *FOCS*, 2011.

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

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

[14] Armando Castañeda and Sergio Rajsbaum. New combinatorial topology upper and lower bounds for renaming. In *PODC '08*, pages 295–304, New York, NY, USA, 2008. ACM.

[15] Bogdan S. Chlebus and Dariusz R. Kowalski. Asynchronous exclusive selection. In *PODC '08*, pages 375–384, New York, NY, USA, 2008. ACM.

[16] 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.

[17] Rui Fan and Nancy Lynch. An O(n log n) lower bound on the cost of mutual exclusion. In *PODC '06*, pages 275–284, New York, NY, USA, 2006. ACM.

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

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

[20] 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.

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

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

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

[24] Prasad Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In *PODC '98*, pages 201–210, New York, NY, USA, 1998. ACM.

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

[26] Yong-Jik Kim and James H. Anderson. A time complexity bound for adaptive mutual exclusion. In *DISC '01*, pages 1–15, London, UK, UK, 2001. Springer-Verlag.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

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