Tight Bounds for Asynchronous Renaming

DAN ALISTARH, Microsoft Research Cambridge
JAMES ASPNES, Yale
KEREN CENSOR-HILLEL, Technion
SETH GILBERT, National University of Singapore
RACHID GUERRAOUI, EPFL

18

This article presents the first tight bounds on the time complexity of shared-memory renaming, a fundamental problem in distributed computing in which a set of processes need to pick distinct identifiers from a small namespace.

We first prove an individual lower bound of $\Omega(k)$ process steps for deterministic renaming into any namespace of size subexponential in k, where k is the number of participants. The bound is tight: it draws an exponential separation between deterministic and randomized solutions, and implies new tight bounds for deterministic concurrent fetch-and-increment counters, queues, and stacks. The proof is based on a new reduction from renaming to another fundamental problem in distributed computing: mutual exclusion. We complement this 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 result applies to randomized algorithms against a strong adversary, and helps derive new global lower bounds for randomized approximate counter implementations, that are tight within logarithmic factors.

On the algorithmic side, we give a protocol that transforms any sorting network into a randomized strong adaptive renaming algorithm, with expected cost equal to the depth of the sorting network. This gives a tight adaptive renaming algorithm with expected step complexity $O(\log k)$, where k is the contention in the current execution. This algorithm is the first to achieve sublinear time, and it is time-optimal as per our randomized lower bound. Finally, we use this renaming protocol to build monotone-consistent counters with logarithmic step complexity and linearizable fetch-and-increment registers with polylogarithmic cost.

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

General Terms: Theory, Algorithms, Performance

Additional Key Words and Phrases: Distributed computing, shared memory, concurrent data structures, renaming, lower bounds

1. INTRODUCTION

The availability of unique names, or identifiers, is a fundamental requirement for distributed computation. Even in settings where unique identifiers such as MAC or IP addresses are available, they often come from a very large namespace, which reduces their usefulness. The renaming problem, in which a set of processes need to pick unique names from a small namespace, is one of the fundamental problems in distributed computing. Intuitively, renaming can be seen as the dual of the consensus problem [Pease et al. 1980]: if solving consensus requires processes to agree on a single value, renaming asks processes to disagree in a constructive way, by returning distinct values from a small space of names.

More precisely, the renaming problem assumes that processes have unique initial names from a large, virtually unbounded namespace, and requires each process to eventually return a name (the *termination* condition), and that the names returned should be *unique* (the *uniqueness* condition). The size of the resulting namespace should be at most $T > 0$, which is given in advance. The namespace size T should only depend on n, the maximum number of participating processes. The *adaptive* version of the renaming problem requires the size of the namespace T to only depend on k, the number of processes actually taking steps in the current execution, also known as the *contention* in the execution. If the size of the namespace matches exactly the number of participating processes, renaming is said to be *strong*, and the namespace is said to be *tight*. Otherwise, renaming is *loose*. Intuitively, a tight namespace is desirable since it minimizes the number of "wasted" names, which are allocated but go unused.

A significant amount of research, for example, Attiya et al. [1990], Bar-Noy and Dolev [1989], Burns and Peterson [1989], Moir and Anderson [1995], Herlihy and Shavit [1999], Afek and Merritt [1999], Attiya and Fouren [2001], Eberly et al. [1998], and Panconesi et al. [1998], has studied the solvability and complexity of renaming in an asynchronous environment. In particular, *tight*, or *strong* deterministic renaming, where the size of the namespace is exactly n, is known to be impossible [Herlihy and Shavit 1999; Castañeda and Rajsbaum 2010]. In fact, $(n + t - 1)$ is the best achievable namespace size when t processes may crash [Castañeda and Rajsbaum 2010, 2012]. The proof of this result required the use of complex techniques [Herlihy and Shavit 1999; Gafni 2009]. This impossibility result can be circumvented through the use of randomization: there exist randomized renaming algorithms that ensure a tight namespace of n names, guaranteeing name uniqueness in all executions and termination with probability 1, for example, Eberly et al. [1998]. Despite considerable research effort on efficient renaming algorithms, for example, Panconesi et al. [1998], Borowsky and Gafni [1993], Moir and Anderson [1995], Moir and Garay [1996], Afek and Merritt [1999], Attiya and Fouren [2001], Eberly et al. [1998], Chlebus and Kowalski [2008], and Afek et al. [1999], prior to our work there have been no time optimality results for shared-memory renaming, either for randomized or deterministic algorithms.

1.1. Overview of the Results

In this article, we present the first tight time-complexity bounds for this problem, both for deterministic and randomized implementations. For deterministic algorithms, we

give a tight *linear* lower bound on renaming into any subexponential namespace.[1]
For randomized algorithms, we give tight *logarithmic* upper and lower bounds on the
time complexity of adaptive renaming. Together, our results give an exponential-time
complexity separation between deterministic and randomized renaming.

Since renaming can be solved trivially using objects with stronger semantics, such as
stacks, queues, or fetch-and-increment counters, our lower bounds also apply to these
widely used concurrent objects. These results improve or match the previously known
lower bounds for these problems (see Figure 1 for an overview). On the other hand,
our renaming algorithms can be extended to obtain new efficient implementations of
other shared objects, such as counters, mutex, test-and-set or fetch-and-increment. (A
summary is given in Figure 2.)

From a technical perspective, this article highlights new connections between renam-
ing and other fundamental objects: sorting networks [Knuth 1998] and mutual exclu-
sion [Dijkstra 1965]. We show that sorting networks can be used to obtain optimal-time
solutions for randomized renaming. In turn, such renaming solutions can be used to
obtain efficient mutual exclusion algorithms. We then proceed by reduction, and derive
a lower bound on renaming from a known lower bound on the time complexity of mu-
tual exclusion [Kim and Anderson 2012]. This result then generalizes to more complex
objects that solve renaming. The lower bound on the time complexity of randomized
renaming follows from a separate information-based argument.

In the following, we describe these contributions in more detail.

1.2. Deterministic Renaming Lower Bound

The main contribution of this article is characterizing the time complexity of the renam-
ing problem in asynchronous shared memory. For deterministic algorithms, we prove
that $\Theta(k)$ process steps is a tight bound for the individual step complexity of adaptive
renaming in a subexponential namespace[2] in the number of participants k. This result,
whose proof can be found in Section 7, extends to *nonadaptive* renaming, to yield a
linear lower bound on the complexity of renaming in a polynomial namespace in n. It
holds for wait-free algorithms using reads, writes, test-and-set, and compare-and-swap
operations, and is matched by various algorithms in the literature, for example, Moir
and Anderson [1995] and Moir and Garay [1996].

We obtain the result by reduction from a lower bound on mutual exclusion. The
proof is structured in two steps. The first step assumes a wait-free algorithm R, re-
naming adaptively into a loose namespace of size $T(k)$, of size subexponential in k. We
transform this algorithm into a *strong adaptive* renaming algorithm $S(R)$, by adding
an $O(\log T(k))$ term to the complexity of the original algorithm R. This first trans-
formation is based on a connection between renaming and sorting networks [Knuth
1998]. (We also exploit this connection to obtain a time-optimal randomized renaming
algorithm.)

The second step in the proof relates adaptive strong renaming to mutual exclusion.
We prove that any strong adaptive renaming algorithm (and thus, also the algorithm
$S(R)$) can be used to solve mutual exclusion with a constant overhead in terms of com-
plexity. This second transformation yields an algorithm $Mutex(S(R))$ that solves mutual
exclusion with worst-case complexity $O(C(R) + \log T(k))$, where $C(R)$ is the worst-case
complexity of the original renaming algorithm R. We now apply a linear lower bound
by Kim and Anderson [2012] on the complexity of adaptive mutual exclusion to obtain

---

[1]This bound is matched by previously known algorithms, for example, Moir and Anderson [1995]. See
Section 4 for a detailed discussion.
[2]More precisely, a subexponential namespace is a namespace of size $o(\alpha^k)$, for any constant $\alpha > 1$.

that the algorithm R must have linear complexity in k as long as the namespace $T(k)$ is a subexponential function in k.

Intuitively, the argument shows that, given k processes that need to pick unique names from a large namespace, there exists a worst-case schedule in which a process executes for $\Omega(k)$ steps, roughly one step for every other process executing in the system. This is somewhat surprising, since it implies that giving the processes more choice for the namespace size does not help in terms of worst-case step complexity: assigning names in a huge namespace, for example, of size $\Theta(k^{100})$, is asymptotically no easier that renaming in a small namespace of size $O(k)$.

The reduction technique implies an even stronger linear lower bound, on the number of *remote memory references* (RMRs) that a process has to perform in a worst-case execution in the cache-coherent shared memory model. In brief, RMRs are a complexity measure that takes into account the number of *cache misses* that a process incurs while running an algorithm, and can be orders of magnitude slower than accesses to local memory on modern multiprocessor architectures.

1.3. Randomized Adaptive Renaming Lower Bound

We complement the deterministic lower bound by also analyzing the time complexity of *randomized* adaptive renaming. More precisely, we analyze the worst-case expected *total* number of steps that processes must perform. We prove a global step complexity lower bound: given any algorithm that renames into a namespace of size ck, with $c \geq 1$, there exists an adversarial strategy that causes the k processes to take $\Omega(k \log(k/c))$ total steps in expectation. This lower bound applies to algorithms using reads, writes, test-and-set and compare-and-swap primitives, and to algorithms that may not terminate with some nonzero probability. This total step complexity lower bound is tight for $c = 1$, that is, for strong adaptive renaming, since it is matched by the renaming network algorithm presented in Section 5.

The same technique implies an $\Omega(k \log(k/c))$ total step complexity lower bound for randomized implementations of *c*-approximate shared counters, that is, counters that return a result that is within a factor of c of the real value. This lower bound is tight within logarithmic factors [Aspnes et al. 2012a], and limits the complexity gain from allowing approximation within constant factors.

Our argument follows the structure of a previous result by Jayanti [1998], which in turn is similar to a lower bound by Cook et al. [1986] on the complexity of computing basic logical operations on PRAM machines. Jayanti proved an $\Omega(\log k)$ lower bound on the expected step complexity of shared counters, queues, and stacks, which also applies to renaming. We generalize his result in two ways: first, we consider *total* step complexity, and thus obtain a stronger $\Omega(\log k)$ lower bound on the *average* worst-case expected step complexity of the problem. Second, our results also apply to loose (approximate) versions of renaming and counting, bounding the benefits of relaxing the object semantics.

1.4. Lower Bounds for Other Objects

Since more complex shared-memory objects such as queues, stacks, or fetch-and-increment counters solve adaptive strong renaming with constant complexity overhead, it follows that the local and global lower bounds stated in the previous two sections apply to these objects as well.

In particular, the deterministic lower bound implies that wait-free deterministic implementations of these objects have linear step complexity in the worst case, suggesting that they do not scale well in terms of worst-case time complexity. The result holds for algorithms that are adaptive (whose complexity depends only on the contention in the execution), or if the algorithms do not assume any bound on the size of the initial

Shared Object	Lower Bound	Type	Matching Algorithms	New Result
Deterministic ck-renaming	$\Omega(k)$	Local	[Moir and Garay 1996]	Yes
	$\Omega(k\log(k/c))$	Global	-	Yes
Randomized ck-renaming	$\Omega(k\log(k/c))$	Global	Section 5	Yes
c-Approximate Counter	$\Omega(k\log(k/c))$	Global	[Aspnes et al. 2012a]	Yes
Fetch-and-Increment	$\Omega(k)$	Local	[Moir and Anderson 1995]	Improves on Fich et al. [2005]
	$\Omega(k\log k)$	Global	Section 5	Improves on Attiya and Hendler [2010]
Queues and Stacks	$\Omega(k)$	Local	[Herlihy 1991]	Improves on Fich et al. [2005]
	$\Omega(k\log k)$	Global	-	Improves on Attiya and Hendler [2010]

Fig. 1.   Summary of the lower bound results and relation to previous work.

namespace of participating processes. (We discuss ways of circumventing the lower bound in Sections 7.3 and 7.4.)

Similarly, the total step complexity lower bound also applies to queues, stacks, and fetch-and-increment out of read-write registers with compare-and-swap operations, giving an $\Omega(k\log k)$ lower bound for exact implementations in executions where k processes participate. This bound is more general since it applies to randomized algorithms as well, and to algorithms that assume names from a small namespace. These lower bounds are matched by several known implementations (please see Figure 1 for a case-by-case description).

1.5. An Algorithm for Strong Randomized Renaming

Our main algorithmic contribution is a time-optimal algorithm for strong adaptive renaming based on a variation of sorting networks [Cormen et al. 2009] which we call *renaming networks*. A renaming network is a sorting network in which all comparators have been replaced with two-process test-and-set objects.

The mechanism behind the algorithm is that each process is assigned a distinct input port, and follows a path through the network determined by leaving each comparator on its top output wire if it wins the test-and-set, and on the bottom output wire otherwise; the output name is the index of the port that it reaches. This construction guarantees that if k processes enter the network on distinct input ports, they will reach the first k output ports, thus returning unique names from 1 to k. The expected step complexity of the algorithm is bounded by the maximum number of comparators between an input port and an output port in the sorting network. There exist sorting networks for which this number is logarithmic in the number of input ports [Ajtai et al. 1983].

The key technical issues are assigning unique input ports to the renaming network, and adapting the network size to work for unbounded values of k (required to obtain an adaptive algorithm). We overcome the first obstacle by noticing that input port assignment can be seen as another instance of renaming, and designing a randomized *loose* renaming algorithm with low complexity, which assigns unique names from 1 to k^c for some constant c, with high probability. We overcome the second issue by introducing a new *adaptive* sorting network, whose size can adapt to the number of processes that access it, and whose complexity remains logarithmic whenever truncated to a finite number of input and output ports.

The resulting algorithm guarantees a tight adaptive namespace with complexity $O(\log k)$, with high probability. This is the first known algorithm to achieve tight adaptive renaming in less than linear time. It improves exponentially on previous strong renaming solutions, which had worst-case complexity at least linear, for example, Alistarh et al. [2010]. It also gives an exponential separation between deterministic and randomized renaming algorithms. The total work lower bound in Section 8 shows that this algorithm is in fact optimal, and that no asymptotic complexity improvements are possible by relaxing namespace size within constant factors. We build on this algorithm to obtain a new counter implementation, a bounded-use fetch-and-increment

Shared Object	Time Complexity
Adaptive Randomized Renaming	$O(\log k)$ steps
m-valued Counters	$O(\log m)$ steps
m-valued Fetch-and-Increment	$O(\log k \log m)$ steps
One-Shot Mutual Exclusion	$O(\log k)$ remote memory references

Fig. 2.   Summary of algorithmic results.

object, and a mutual exclusion algorithm with logarithmic-time complexity. The algorithmic results are summarized in Figure 2.

1.6. Roadmap

The article is divided in three parts. (Note that the technical presentation has a slightly different structure than the introduction.) The first part gives the background, outlining the model (Section 2), problem statements (Section 3), and an overview of related work (Section 4). In the second part (Sections 5 and 6), we present a time-optimal randomized renaming algorithm and its applications to counting objects. We focus on lower bounds in part three. We give the deterministic lower bound and its extensions to other objects in Section 7. We then present the proof of the global step complexity lower bound in Section 8, with applications to renaming and counting. We summarize the results and present an overview of open questions in Section 9.

2. SYSTEM MODEL

In this section, we introduce the system model for which our algorithms and lower bounds are designed. We also describe the cost measures under which we will analyze algorithms. In brief, the model we consider is the standard *asynchronous shared memory* model [Attiya and Welch 1998; Lynch 1996] in which processes execute without bounds on their relative execution speed, in the presence of crash failures, communicating through operations on registers.

2.1. The Asynchronous Shared Memory Model

2.1.1. Model Overview. We consider the standard asynchronous shared-memory model, in which n processes $p_1, \ldots, p_n$ communicate through operations on shared multiwriter multireader atomic registers. We will denote by k the *contention* in an execution, that is, the actual number of processes that take steps in the execution.

Processes follow an algorithm, which is composed of *instructions*. Each instruction consists of some local computation, which may include an arbitrary number of local coin flips, and one shared memory operation, such as a read or write to a register, which we call a shared-memory *step*. A number of $t < n$ processes may fail by crashing. (Throughout this article, we assume that the upper bound t on the number of failures is $n-1$.) A *failed* process does not execute any further instructions. A process that does not crash during an execution is *correct*.

The order in which the processes execute shared-memory steps and their crashes are controlled by a *scheduler*, which we model as an *adversary*. More precisely, we allow the adversary to observe the state of all processes, including local coin flips, whenever scheduling the next step. This type of adversary is known as the *strong* adversary.

In the following, we define these terms formally.

2.1.2. Processes, Algorithms, and Shared Objects. A *process* is a sequential unit of computation, created by an application when necessary. For simplicity, we will assume that processes are created at the beginning of each execution, and each executes steps from its algorithm when scheduled. Thus, we denote by $\Pi = \{p_1, p_2, \ldots, p_n\}$ the set of all processes that may execute an algorithm. Accordingly, n will be the total number of

processes that may execute an algorithm. On the other hand, we will denote by k the *actual* number of processes that execute instructions in the execution. Consequently, we have the relation $k \leq n$.

Initially, each process p_i is assigned a unique initial identifier id_i, which, for simplicity, is an integer. We will assume that the space of initial identifiers is of infinite size. This models the fact that, in real systems, processes may use identifiers from a very large space, such as the space of UNIX process identifiers, or the set of all IP addresses.

Each process executes an algorithm assigned to it by the application. An algorithm is a series of *instructions*. A process' next instruction is always determined by its state and by the algorithm. The difference between deterministic and randomized algorithms is that, in the latter case, the state may contain the results of random coin flips performed by the process.

Processes perform local computation, as well as execute operations on *shared objects*. Each operation is described by an *invocation*, and a *response*. For example, the operation

$$val \leftarrow R.\mathsf{read}(\,)$$

reads the contents of shared object R, in this case a register. The response of the read operation is stored in the local variable val. If process p invokes an operation on object X, we say that it *accesses X*. Note that an operation invocation may not necessarily be followed by a response event. Such an operation is called a *pending* operation.

2.1.3. Randomization. Some of the algorithms we present are *randomized*, in that the processes' actions may depend on the outcomes of *local* coin flips. In general, we use randomization in algorithms in order to assign probability weight to executions, and avoid the worst-case executions with high probability.

Processes may perform local coin flips by calling a local function coin, which takes two integer parameters a and b with $a \leq b$, and returns an integer x with $a \leq x \leq b$ chosen uniformly at random. For example, the call $\mathsf{coin}(0, 1)$ will return 0 with probability $1/2$, and 1 with probability $1/2$.

2.1.4. Concurrent Executions and the Adversarial Scheduler. An execution is a sequence of operations performed by a set of processes. In order to represent executions, we will assume discrete time, where at every unit there is only one active process. In a time unit, the active process can perform any number of local computations or local coin flips, and then issue an *event* or execute a *step*.

Events are local to each process, that is, are always received by the process issuing the event, and are used to mark the time when a process starts and stops invoking an implementation of a shared object as part of its algorithm. (For example, the time when a process calls a procedure implementing a lower-level shared object X, and the time when it returns from this procedure.) Processes are sequential, that is, in each step a process may execute at most one operation on at most one object. In particular, the *return* event is used by the process to return from the algorithm.

A *step* is an execution of an operation on a base object, which comprises the invocation and the subsequent response (therefore, every operation on a base object takes at most one unit of time to execute). Whenever a process p_i becomes active (as decided by the scheduler), p_i executes an event or a step. We assume that the scheduler has access to the results of any local computation or local coin flips leading to the step, before choosing to schedule it. It may be possible that a process does not have anything to execute, for example, if it terminates its algorithm, in which case it executes an empty no-op step.

The order in which processes take steps and issue events is determined by an external abstraction called a *scheduler*, over which processes do not have control. In the following, we will consider the scheduler as an *adversary*, whose goal is to maximize

the cost of the protocol (in this article, we focus on number of steps as a cost measure). Thus, we will use the terms *adversary* and *scheduler* interchangeably. The adversary controls the *schedule*, which is a (possibly infinite) sequence of process identifiers. If process p_i is in position τ of the sequence, then this implies that p_i is active at time τ. The adversary has the freedom to schedule any interleaving that complies with the given model. In this article, we assume an asynchronous model; therefore, the adversary may schedule any interleaving of process steps.

Consequently, an execution is a sequence of all events and steps issued by processes in a given run of an implementation. Every execution has an associated schedule, which yields the order in which processes are active in the execution. For deterministic algorithms, the schedule completely determines the execution.

For randomized algorithms, notice that different assumptions on the relation between the scheduler and the random coin flips that processes perform during an execution may lead to different results. In this article, we will assume that the adversary controlling the schedule is a *strong* adversary, that observes the results of the local coin flips, together with the state of all processes, before scheduling the next process step (in particular, the interleaving of process steps may depend on the result of their coin flips).

This is the standard adversarial model for randomized distributed algorithms, which reflects the fact that the speed of a process may be influenced by the results of random coin flips that the process performs. On the one hand, it is the strongest "reasonable" adversarial model, since a stronger adversary would have to be aware of the results of coin flips that the processes perform in the future. On the other hand, it encompasses weaker adversarial models, such as the *oblivious* adversary, for example, [Alistarh and Aspnes 2011], which fixes the scheduling and failure pattern independently of the processes' coin flips.

2.2. Asynchrony and Wait-Freedom

In this article, we focus on *asynchronous* shared-memory systems. In such systems, the time delay between two consecutive events of any process may be arbitrary, that is, there are no assumptions on the relative speed of processes. This models the fact that, in general-purpose systems, processes may be preempted or otherwise delayed for arbitrary periods of time. While real-world systems may not be entirely asynchronous, proving algorithms correct in the asynchronous model ensures that they will be correct in any system in which delays are bounded. In this setting, an implementation is *wait-free* if any process invoking an operation also returns within some finite number of its own steps. In this article, we focus on wait-free algorithms.

2.3. Complexity Measures

We measure complexity in terms of process steps, where each shared-memory operation is counted as one step. Thus, the (individual) *step complexity* of an algorithm is the worst-case number of steps that a single process may have to perform in order to return from an algorithm, including invocations to lower-level shared objects. The *total* step complexity is the total number of shared memory operations that all participating processes perform during an execution. For randomized algorithms, we will analyze the worst-case *expected* number of steps that a process may perform during an execution as a consequence of the adversarial scheduler, or give more precise probability bounds for the number of steps performed during an execution.

For the lower bound in Section 7, we will use a stronger measure of complexity, by counting the number of *remote memory references* (RMRs). In cache-coherent (CC) shared memory, each process maintains local copies of shared variables inside its cache. The consistency of the cache among processes is ensured by a coherence protocol. A

variable is *remote* to a process if its cache contains a copy of the variable whose value is out of date (or if the cache contains no copy of the variable); otherwise, the variable is *local*. A process step is *local* if it accesses a local variable. Otherwise, the step is a *remote memory reference* (RMR). A similar definition exists for the distributed shared memory (DSM) model. Notice that, since each RMR implies a distinct process step, RMR complexity is always a lower bound on step complexity.

2.4. Linearizability

Linearizability [Herlihy and Wing 1990] is a correctness condition for shared object implementations. Intuitively, an implementation is linearizable (or atomic) if every shared memory operation should appear to the processes as if it was executed instantaneously at some single and unique point in time between its invocation and its response. This notion helps keep the algorithms simple, by eliminating technical details and providing a clean interface. The semantics of atomic objects can be described by using their sequential behavior, that is, by giving their sequential specification.

More precisely, an implementation of an object O is *linearizable* if, for every execution, there exists a total order over all the complete process operations operations together with a subset of the incomplete process operations such that every operation is immediately (atomically) followed by a response, and the sequence of operations given by that total order is consistent with a sequential execution of the object O.

Linearizability and Randomization. Recent results by Golab et al. [2011] show that linearizability is not a sufficient correctness condition when randomization is employed. More precisely, they show that the adversary can gain extra power whenever a randomized algorithm uses other (deterministic or randomized) linearizable implementations as subalgorithms. In this article, we circumvent this technical issue by avoiding the use of linearizability as a correctness condition when employing subalgorithms: instead, we isolate a set of invariants whenever we use a known implementation as a subalgorithm.

3. PROBLEM STATEMENTS AND SHARED OBJECTS

We now present the definitions and sequential specifications of the problems and objects considered in this article.

3.1. Renaming

The *renaming problem*, introduced in Attiya et al. [1990], is defined as follows. Each of the n processes has initially a distinct identifier id_i taken from a domain of potentially unbounded size M, and should return an output name o_i from a smaller domain. (Note that the index i is only used for description purposes, and is not known to the processes.) Given an integer T, an object ensuring *deterministic* renaming into a target namespace of size T, also called a T-*renaming* object, guarantees the following properties.

(1) *Termination*. In every execution, every correct process returns a name.
(2) *Namespace Size*. Every name returned is from 1 to T.
(3) *Uniqueness*. Every two names returned are distinct.

The *randomized renaming* problem relaxes the termination condition, ensuring *randomized termination*: with probability 1, every correct process returns a name. The other two properties stay the same.

The domain of values returned, which we call the *target namespace*, is of size T. In the classical renaming problem [Attiya et al. 1990], the parameter T may not depend on the range of the original names. On the other hand, it may depend on the parameter n and on the number of possible faults t.

For *adaptive* renaming, the size of the resulting namespace, and the complexity of the algorithm, should only depend on the number of participating processes k in the current execution. In some instances of the problem, processes are assumed not to know the maximum number of processes n, whereas in other instances an upper bound on n is provided. (In this article, we consider the slightly harder version in which the upper bound on n is not provided.)

If the size of the namespace matches exactly the number of participating processes, then we say that the target namespace is *tight*. Consequently, the strong renaming problem requires that the processes obtain unique names from 1 to n, that is, $T = n$. The *strong adaptive* renaming problem requires that k participating processes obtain consecutive names $1, 2, \ldots, k$. Thus, strong adaptive renaming is the version of the problem with the largest number of constraints. To distinguish the classical renaming problem from the adaptive version, we will denote the classical version, where n is given and complexity and namespace depend on n, as the *nonadaptive* renaming problem.

3.2. Registers

The simplest base object we will use is the *register*. The sequential specification of the object exports two operations:

—read( ), which returns the current state (value) of the object,
—write(v), which changes the state of the object to value v, and returns success.

If a process p_i executes a read operation on a register R, we say that p_i *reads* R. Similarly, we say that p_i *writes* value v to R if it invokes a write(v) operation on register R.

3.3. Test-and-Set and Compare-and-Swap

The *test-and-set* object, whose sequential specification is given in Figure 3, can be seen as a tournament object for n processes. In brief, the object has initial value 0, and supports a single test-and-set operation, which atomically sets the value of the object to 1, returning the value of the object before the invocation. Notice that at most one process may *win* the object by returning the initial value 0, while all other processes *lose* the test-and-set by returning 1. A key property is that no losing test-and-set operation may return before the winning operation is invoked.

More precisely, a correct deterministic implementation of a single-use test-and-set object ensures the following properties.

(1) (Validity.) Each participating process may return one of two indications: 0, or 1.
(2) (Termination.) Each process accessing the object eventually returns or crashes.
(3) (Linearization.) Each execution has a linearization order $\mathcal{L}$ in which each invocation of test-and-set is immediately followed by a response (i.e., is atomic), such that the first response is either 0 or the caller crashes, and no return value of 1 can be followed by a return value of 0.
(4) (Uniqueness.) At most one process may return 0.

For *randomized* test-and-set, the *termination* condition is replaced by the following *randomized termination* property: with probability 1, each process accessing the object eventually returns or crashes. The other requirements stay the same.

In this article, we will use a randomized two-process test-and-set implementation by Tromp and Vitányi [2002] as a basis for our algorithms. Their algorithm ensures the following properties, whose proofs can be found in their paper.

THEOREM 3.1 [TROMP AND VITÁNYI 2002]. *The two-process test-and-set implementation of Tromp and Vitányi [2002] ensures the following properties.*

Variable:;
Value, a binary atomic register,
initially 0;

procedure test-and-set();
  if *Value* = 0 **then**
  | *Value* ← 1;
  | **return** 0;
  end
  else
  | **return** 1;
  end

Fig. 3. Sequential specification of a one-shot test-and-set object.

Variable:;;
V, a register, with initial value ⊥;
procedure compare-and-swap(*oldV*, *newV*);
  s ← *V*;
  if *oldV* = *s* **then**
  | *V* ← *newV*;
  | **return** *s*;
  end
  else
  | **return** *s*;
  end

Fig. 4. Sequential specification of the compare-and-swap object.

—*(Single Winner) In any execution, at most one process may return* 0.
—*(Winner-Loser Ordering) Given any* test-and-set *operation* τ *that returns* 1*, there exists another* test-and-set *operation* w*, starting before* τ *returns, such that* w *either returns* 0 *or does not complete.*
—*(Probabilistic Termination) Operations by correct processes terminate with probability* 1.
—*(Complexity) The algorithm has expected constant read-write step complexity. For a fixed constant* α ≥ 1*, and for any integer* ℓ ≥ 1*, the probability that a process performs more than* α log ℓ *read and write steps while running the algorithm is at most* $1/\ell^2$.

The *compare-and-swap* object can be seen a generalization of the test-and-set object, whose underlying register supports multiple values (as opposed to only 0 and 1). Its sequential specification is presented in Figure 4. More precisely, a compare-and-swap object exports the following operations:

—read and write, having the same semantics as for registers,
—compare-and-swap(*oldV*, *newV*), which compares the state *s* of the object to the value *oldV*, and either (1) changes the state of the object to *newV* and returns *oldV* if *s* = *oldV*, or (b) returns the state *s* if *s* ≠ *oldV*.

Notice that the compare-and-swap object can be seen as an augmented register, which also supports the conditional compare-and-swap operation. Also note that it is trivial to implement a test-and-set object from a compare-and-swap object.

3.4. Counter and Max-Register Objects

A *counter* object has initial state 0, and supports operations increment and read, with the following semantics:

—read( ), which returns the current state (value) of the object,
—increment( ), which changes *v*, the current value of the object to *v* + 1, and returns success.

A *decrementable* counter has the same semantics as a counter, but offers an additional decrement( ) operation, which changes the value *v* of the object to *v* − 1, and returns success.

A *fetch-and-increment* object supports a single operation fetch-and-inc, which changes the value *v* of the object to *v* + 1, and returns value *v*.

The *max-register* is a shared object maintaining a value *V*, initially 0, which records the highest value ever written to it. The *max-register* defines a default maximal value

$v_{\max}$ which it may store. Aspnes et al. [2012a] gave an implementation of a max-register with logarithmic complexity in $v_{\max}$.

THEOREM 3.2 [ASPNES ET AL. 2012a]. *There exists a linearizable, deterministic, wait-free max-register construction for n processes where each operation has cost $O(\min(n, \log v_{\max}))$.*

3.5. Mutual Exclusion

The goal of the mutual exclusion (mutex) problem is to allocate a single, indivisible, non-shareable resource among n processes. A process with access to the resource is said to be in the *critical section*. When a user is not involved with the resource, it is said to be in the *remainder section*. In order to gain admittance to the critical section, a user executes an *entry section*; after it is done with the resource, it executes an *exit section*. Each of these sections can be associated with a partitioning of the code that the process is executing.

Each process cycles through these sections in the order: remainder, entry, critical, and exit. Thus, a process that wants to enter the critical section first executes the entry section; after that, it enters the critical section, after which it executes the exit section, returning to the remainder section. We assume that in all executions, each process executes this section pattern infinitely many times. For simplicity, we assume that the code in the remainder section is trivial, and every time the process is in this section, it immediately enters the entry section. An execution is *admissible* if for every process p_i, either p_i takes an infinite number of steps, or p_i's execution ends in the remainder section. A *configuration* at a time τ is given by the code section for each of the processes at time τ.

An algorithm solves mutual exclusion with no deadlock if the following hold. We adopt the definition of Attiya and Welch [1998].

—*Mutual exclusion*. In every configuration of every execution, at most one process is in the critical section.
—*No deadlock*. In every admissible execution, if some process is in the entry section in a configuration, then there is a later configuration in which some process is in the critical section.
—*No lockout (Starvation-free)*. In every admissible execution, if some process is in the entry section in a configuration, then there is a later configuration in which *the same* process is in the critical section.
—*Unobstructed exit*. In every execution, every process returns from the exit section in a finite number of steps.

In this article, we focus on shared-memory mutual exclusion algorithms. As for renaming, there exists a distinction between adaptive and nonadaptive solutions. A classical, nonadaptive, mutual excusion algorithm is an algorithm whose complexity depends on n, the maximum number of processes that may participate in the execution, which is assumed to be known by the processes at the beginning of the execution. On the other hand, an *adaptive* mutual exclusion algorithm is an algorithm whose complexity may only depend on the number of processes k participating in the current execution.

3.6. Queues and Stacks

The *queue* is a data structure which maintains a set of elements with first-in-first-out (FIFO) semantics. More precisely, the state of a queue can be described as an array $[x_0 = \bot, x_1, \ldots, x_m]$. A queue is *empty* if its state is $[\bot]$.

Assume a queue in state $[x_0 = \bot, x_1, \ldots, x_m]$. The sequential specification of a queue supports two operations.

—The Enqueue(v) operation changes the state of the stack to $[x_0 = \bot, x_1, \ldots, x_m, v]$, returning success. We assume $v \neq \bot$.

—The Dequeue() operation returns the "oldest" element in the queue. More precisely, if $m \geq 1$, then the Dequeue() operation returns x_1 and changes the state of the stack to $[x_0 = \bot, x_2, \ldots, x_m]$. Otherwise, the operation returns $\bot$.

The *stack* is a data structure which maintains a set of elements with last-in-first-out (LIFO) semantics. More precisely, the state of a stack can be described as an array $[x_0 = \bot, x_1, \ldots, x_m]$, where the last element x_m is the *top* of the stack. A stack is *empty* if its state is $[\bot]$.

Assume a stack in state $[x_0 = \bot, x_1, \ldots, x_m]$. The sequential specification of a stack supports two operations.

—The Push(v) operation changes the state of the stack to $[x_0 = \bot, x_1, \ldots, x_m, v]$, returning success. We assume $v \neq \bot$.
—The Pop() operation returns the top of the stack. If $x_m \neq \bot$, then the Pop() operation also changes the state of the stack to $[x_0 = \bot, x_1, \ldots, x_{m-1}]$.

4. RELATED WORK

In this section, we provide an overview of research on renaming and related data structures in the shared memory and message-passing models.

4.1. Renaming

The renaming problem, defined in Section 3.1, was introduced by Attiya et al. [1990], in the asynchronous message-passing model. Their article presented a nonadaptive algorithm that achieves $(2n - 1)$ names in the presence of $t < n/2$ faults, and showed that a tight namespace of n names cannot be achieved in an asynchronous system with crash failures. It also introduced and studied a version of the problem called *order-preserving* renaming, in which the final names have to respect the relative order of the initial names.

Renaming has been studied in a variety of models and under various timing assumptions. For synchronous message-passing systems, Chaudhuri et al. [1999] gave a wait-free algorithm for strong renaming in $O(\log n)$ rounds of communication, and proved that this upper bound is asymptotically tight if the number of process failures is $t \leq n - 1$ and the algorithm is *comparison-based*, that is, two processes may distinguish their states only through comparison operations. Attiya and Djerassi-Shintel [2001] studied the complexity of renaming in a semi-synchronous message-passing system, subject to timing faults. They obtained a strong renaming algorithm with $O(\log n)$ rounds of broadcast and proved a $\Omega(\log n)$ time lower bound when algorithms are comparison-based or when the initial namespace is large enough compared to n. Both these algorithms can be made adaptive, to obtain a running time of $O(\log k)$. Okun [2010] presented a strong renaming algorithm that is also *order-preserving*, with $O(\log n)$ time complexity. The algorithm exploits a new connection between renaming and approximate agreement [Fekete 1990]. Recently, Alistarh et al. [2012] analyzed Okun's algorithm and showed that it is also *early-deciding*, that is, its running time can adapt to the number of failures $f \leq n - 1$ in the execution. In particular, they showed that the algorithm terminates in a *constant* number of rounds, if $f < \sqrt{n}$, and in $O(\log f)$ rounds otherwise.

The first shared-memory renaming algorithm was given by Bar-Noy and Dolev [1989], who ported the synchronous message-passing algorithm of Attiya et al. [1990] to use only reads and writes. They obtained an algorithm with namespace size $(k^2 + k)/2$ that uses $O(n^2)$ steps per operation, and an algorithm with a namespace size of $(2k - 1)$ using $O(n \cdot 4^n)$ steps per operation.

Early work on lower bounds focused on the size of the namespace that can be achieved using only reads and writes. Burns and Peterson [1989] proved that, for any

$T(n) < 2n - 1$, *long-lived renaming*[3] in a namespace of size $T(n)$ is impossible in asynchronous shared memory using reads and writes. They also gave the first long-lived $(2n - 1)$-renaming algorithm. (However, the complexity of this algorithm depends on the size of the initial namespace, which is not allowed by the original problem specification [Attiya et al. 1990].) In a landmark paper, Herlihy and Shavit [1999] used algebraic topology to show that there exist values of n for which $(2n - 2)$-renaming is impossible. Recently, Castañeda and Rajsbaum [2010, 2012] proved that if n is a prime power, then target namespace size $T(n) \geq 2n - 1$ is necessary, and, otherwise, there exists an algorithm with $2n - 2$ namespace size.

A parallel line of work [Lipton and Park 1990; Kutten et al. 2000] studied *anonymous* renaming, where processes do not have initial identifiers and start in identical state. In this case, renaming cannot be achieved with probability 1 using only reads and writes, since one cannot distinguish between processes in the same state, and thus two processes may always decide on the same name with nonzero probability.

Afek and Merritt [1999] presented an adaptive read-write renaming algorithm with optimal namespace of size $(2k - 1)$, and $O(k^2)$ step complexity. Attiya and Fouren [2001] gave an adaptive $(6k - 1)$-renaming algorithm with $O(k \log k)$ step complexity. Chlebus and Kowalski [2008] gave an adaptive $(8k - \log k - 1)$-renaming algorithm with $O(k)$ step complexity. For *long-lived* adaptive renaming, there exist implementations with $O(k^2)$ time complexity for renaming into a namespace of size $O(k^2)$, for example, Afek et al. [1999]. The fastest such algorithm with optimal $(2k - 1)$ namespace size has $O(k^4)$ step complexity [Attiya and Fouren 2001].

The time lower bound in Section 7 shows that linear-time deterministic algorithms are in fact time optimal (since they ensure namespaces of polynomial size). On the other hand, the existence of a deterministic read-write algorithm which achieves both an optimal namespace and linear-time complexity is an open problem.

The relation between renaming and stronger primitives such as fetch-and-increment or test-and-set was investigated by Moir and Anderson [1995]. Fetch-and-increment can be used to solve renaming trivially, since each process can return the result of the operation plus 1 as its new name. Renaming can be solved by using an array of test-and-set objects, where each process accesses test-and-set objects until winning the first one. The process then returns the index of the test-and-set object that it has acquired. Moir and Anderson [1995] also present implementations of renaming from registers supporting set-first-zero and bitwise-and operations. In this article, the authors also notice the fact that adaptive tight renaming can solve mutual exclusion. (This connection is also mentioned in Attiya and Bortnikov [2002].) Using load-linked and store-conditional primitives, Brodsky et al. [2006] gave a linear-time algorithm with a tight namespace. (Their paper also presents an efficient synchronous shared-memory algorithm.)

Randomization is a natural approach for obtaining names, since random coin flips can be used to "balance" the processes' choices. A trivial solution when n is known is to have processes try out random names from 1 to n^2. Name uniqueness can be validated using deterministic splitter objects [Anderson and Moir 1997], and the algorithm uses a constant number of steps in expectation, since, by the birthday paradox, the probability of collision is very small. The feasibility of randomized renaming in asynchronous shared memory was first considered by Panconesi et al. [1998]. They presented a nonadaptive wait-free solution with a namespace of size $n(1 + \epsilon)$ for $\epsilon > 0$ constant, with expected $O(M \log^2 n)$ running time, where M is the size of the initial namespace.

---

[3]The *long-lived* version of renaming allows processes to release names as well as to acquire them.

A second paper to analyze randomized renaming was by Eberly et al. [1998]. The authors obtain a *strong* nonadaptive renaming algorithm based on the randomized wait-free test-and-set implementation of Afek et al. [1992]. Their algorithm is long-lived, and is shown to have amortized step complexity $O(n \log n)$. The average-case total step complexity is $\Theta(n^3)$.

A paper by Alistarh et al. [2010] generalized the approach by Panconesi et al. [1998] by introducing a new, *adaptive* test-and-set implementation with logarithmic step complexity, and a new strategy for the processes to pick which test-and-set to compete in: each process chooses a test-and-set between 1 and n at random. The authors prove that this approach results in a nonadaptive tight algorithm with $O(n \operatorname{polylog} n)$ total step complexity.[4] (However, in this algorithm, individual processes may still perform a linear number of accesses.) A modified version of this approach generates an *adaptive* algorithm with similar complexity, which ensures a loose namespace of size $(1 + \epsilon)k$, for $\epsilon > 0$ constant.

The randomized algorithm presented in this article first appeared in Alistarh et al. [2012]. The renaming network algorithm is the first algorithm to achieve strong adaptive renaming in sub-linear time, improving exponentially on the time complexity of previous solutions. The lower bound in Section 8 shows that this algorithm is in fact time-optimal. The fact that any sorting network can be used as a counting network when only one process enters on each wire was observed by Attiya et al. [1995] to follow from earlier results of Aspnes et al. [1994]; this is equivalent to our use of sorting networks for nonadaptive renaming in Section 5.1.1. The lower bounds in this article first appeared in Alistarh et al. [2011b].

4.2. Counting Data Structures

Many multiprocessor coordination tasks can be expressed as counting problems, where processes assign values from a given range. Thus, there has been considerable work on counting data structures over the last few decades. *Counting networks* [Aspnes et al. 1994] are an example of such data structures, where processes interact with a counter by traversing a network of *balancer* objects.[5] Efficient counting networks with $O(\operatorname{polylog} n)$ complexity are known [Aspnes et al. 1994]. Counting networks are similar to the renaming networks presented in Section 5.1.1: however, the aim of a counting network is to balance the number of processes exiting on the output ports, whereas renaming networks ensure that no two processes reach the same output port. As a consequence, the structure and applications of counting networks are in general different than those of renaming networks.

Another well-studied object is the *counter*. A linear-time deterministic atomic counter implementation follows from the atomic snapshot construction of Afek et al. [1993]. Jayanti et al. [2000] proved $\Omega(n)$ space and time lower bounds for deterministic counter implementations if processes may access the object multiple times, while Jayanti [1998] gave $\Omega(\log n)$ time lower bounds for counter objects using reads, writes, or load-linked/store-conditional operations.

A recent result by Aspnes et al. [2012a] leveraged the fact that objects may be accessed for a limited number of times to give an *m*-valued max register implementation with $O(\min(n, \log m))$ time complexity, and a $O(\min(n, \log m \log n))$ upper bound for deterministic wait-free counters with maximal value m. Recently [Aspnes et al. 2012b], this technique was generalized to obtain atomic snapshots with $O(\log^2 b \cdot \log n)$ time

---

[4]In the following, by $\operatorname{polylog} n$, we denote $\log^c n$, for some integer $c \geq 1$.
[5]Intuitively, a balancer acts as a *toggle* mechanism. It has two inputs and two outputs; processes enter the object on its inputs, and the balancer alternates sending processes to its top and bottom output wires.
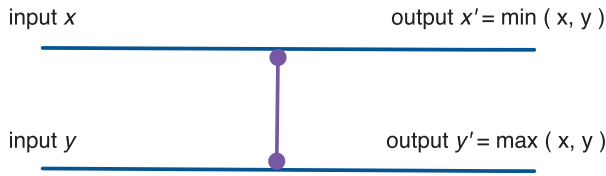
input *x*   output *x′* = min (x, y)

input *y*   output *y′* = max (x, y)

Fig. 5. Structure of a comparator.

complexity for a scan, and $O(\log b)$ complexity for an update, where b is the number of update operations performed on the object.

5. ADAPTIVE STRONG RENAMING IN LOGARITHMIC EXPECTED TIME

In this section, we present an algorithm for *strong adaptive* renaming, which ensures a namespace that exactly matches the number of participants k. The algorithm has expected step complexity $O(\log k)$, which is optimal, as we show in Section 8.

Renaming Networks. The key ingredient behind the algorithm is a connection between renaming and *sorting networks*, a data structure used for sorting sequences of numbers in parallel. In brief, we start from a sorting network, and replace the comparator objects with two-process test-and-set objects, to obtain an object we call a *renaming network*. The algorithm works as follows: each process is assigned a unique input port, and follows a path through the network determined by leaving each two-process test-and-set on its higher output wire if it wins the test-and-set, and on its lower output wire if it loses. The output name is the index (from top to bottom) of the output port it reaches.

There are two major obstacles to turning this idea into a strong adaptive renaming algorithm. The first is that this construction is not adaptive. Since the step complexity of running the renaming network depends on the number of input ports assigned, then, if we simply use the processes' initial names to assign input ports, we could obtain an algorithm with unbounded worst-case step complexity, since the space of initial identifiers is potentially unbounded. The second obstacle is that a regular sorting network construction has a fixed number of input and output ports, therefore the construction would not adapt to the contention k. Since we would like to avoid assuming any bound on the contention, we need to build a sorting network that "extends" its size as the number of participating processes increases.

In the following, we show how to overcome these problems, and obtain a strong adaptive renaming algorithm with complexity $O(\log k)$, with high probability in k.[6] In the next section, we use this construction to obtain randomized implementations of counter and fetch-and-increment objects.

5.1. Renaming using a Sorting Network

We now give a strong renaming algorithm based on a sorting network. For simplicity, we describe the solution in the case where the bound on the size of the initial namespace, M, is finite and known. We circumvent this limitation in Section 5.2.

5.1.1. Renaming Networks. We start from an arbitrary sorting network with M input and output ports, in which we replace the comparators with two-process test-and-set objects. The structure of a comparator is given in Figure 5 (please see standard texts,

---

[6]Notice that, if the contention k is small, the failure probability $O(1/k^c)$ with $c \geq 2$ constant may be nonnegligible. In this case, the failure probability can be made to depend on the parameter n at the cost of a multiplicative $\Theta(\log n)$ factor in the running time of the algorithm.

Shared:;
Renaming network R;

procedure rename(v_i);
    $w \leftarrow$ input wire corresponding to v_i in R;
    while *w is not an output wire* **do**
        $T \leftarrow$ next test-and-set on wire w of R;
        $res \leftarrow T.\text{test-and-set}(\ )$;
        if $res = 0$ **then**
            $w \leftarrow$ output wire x' of T;
        end
        else
            $w \leftarrow$ output wire y' of T;
        end
    end
    return $w.index$;

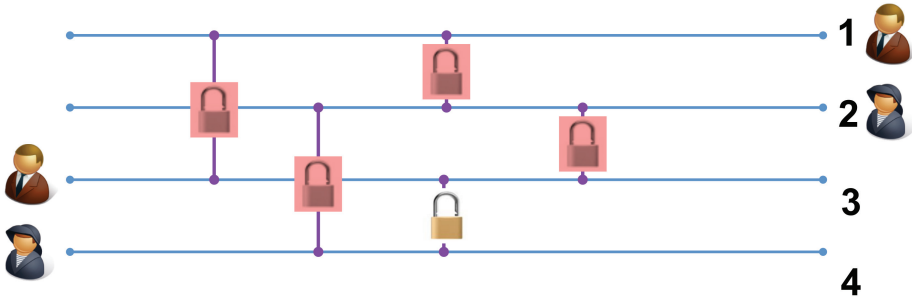Fig. 6. Pseudocode for executing a renaming network.

Fig. 7. Execution of a renaming network. The two processes start at arbitrary distinct input ports, and proceed through the network until reaching an output port. The two-process test-and-set objects are depicted as locks. A two process test-and-set object is highlighted if it has been won during the execution. The execution depicted is one in which processes proceed sequentially (the upper process first executes to completion, then the lower process executes). The two processes reached output ports 1 and 2, even though they started at arbitrary input ports.

e.g. Cormen et al. [2009], for background on sorting networks). The two-process test-and-set objects maintain the input ports x, y and the output ports x', y'. We call this object a *renaming network*.

We assume that each participating process p_i has a unique initial value v_i from 1 to M. (These values can be the initial names of the processes, or names obtained from another renaming algorithm, as described in Section 5.2.) Also part of the process's algorithm is the blueprint of a renaming network with M input ports, which is the same for all participants.

We use the renaming network to solve adaptive tight renaming as follows. (Please see Figure 6 for the pseudocode.) Each participating process enters the execution on the input wire in the sorting network corresponding to its unique initial value v_i. The process competes in two-process test-and-set instances as follows: if the process returns 0 (wins) a two-process test-and-set, then it moves "up" in the network, that is, follows output port x' of the test-and-set; otherwise, it moves "down," that is, follows output port y'. Each process continues until it reaches an output port b_ℓ. The process returns the index ℓ of the output port b_ℓ as its output value. See Figure 7 for a simple illustration of a renaming network execution.

Test-and-Set. In this section, the test-and-set objects used as comparators are implemented using the algorithm of Tromp and Vitányi [2002]; in Section 7, we will assume hardware implementations of test-and-set. This distinction is only important when computing the complexity of the construction, and does not affect its correctness.

5.1.2. Renaming Network Analysis. In the following, we show that the renaming network construction solves adaptive strong renaming, that is, that processes return values between 1 and k, the total contention in the execution, as long as the size of the initial namespace is bounded by M.

THEOREM 5.1 (RENAMING NETWORK CONSTRUCTION). *Whenever starting from a correct sorting network, the renaming network construction solves strong adaptive renaming, with the same progress property as the test-and-set objects used. If the sorting network has depth d (defined here), then each process will perform $O(d)$ test-and-set operations before returning from the renaming network.*

PROOF. First, we prove that the renaming network is *well formed*, that is, that no two processes may access the same port of a two-process test-and-set object.

CLAIM 1. *No two processes may access the same port of a two-process test-and-set object.*

PROOF. Recall that each renaming network is obtained from a sorting network. Therefore, for any renaming network, we can maintain the standard definitions of network and wire depth as for a sorting network [Cormen et al. 2009]. In particular, the depth of a wire is defined as follows. An input wire has depth 0. A test-and-set that has two input wires with depths d_x and d_y will have depth $\max(d_x, d_y) + 1$. A wire in the network has depth equal to the depth of the test-and-set from which it originates. Because there can be no cycles of test-and-sets in a renaming network, this notion is well-defined. The depth of a network is the maximum depth of an output wire.

The claim is equivalent to proving that no two processes may occupy the same wire in an execution of the network. We prove this by induction on the depth of the current wire. The base case, when the depth is 0, that is, we are examining an input wire, follows from the initial assumption that the initial values v_i of the processes are unique, hence no two processes may join the same input port.

Assume that the claim holds for all wires of depth $d \geq 0$. We prove that it holds for any wire of depth $d + 1$. Notice that the depth of a wire may only increase when passing through a two-process test-and-set object. Consider an arbitrary two-process test-and-set object, with two wires of depth at most d as inputs, and two wires of depth $d + 1$ as outputs. By the induction hypothesis, the test-and-set is well formed in all executions, since there may be at most two processes accessing it in any execution. By the specification of test-and-set, it follows that, in any execution, there can be at most one process returning 0 from the object, and at most one process returning 1 from the object. Therefore, there can be at most one process on either output wire, and the induction step holds. This completes the proof of this claim. □

Termination follows since the base sorting network has finite depth and, by definition, contains no cycles. Therefore, the renaming network has the same termination guarantees as the two-process test-and-set algorithm we use. In particular, if we use the two-process test-and-set implementation of Tromp and Vitányi [2002], the network guarantees termination with probability 1. We prove name uniqueness and namespace tightness by ensuring the following claim.

CLAIM 2. *The renaming network construction ensures that no two processes return the same output, and that the processes return values between* 1 *and k, the total contention in the execution.*

The proof is based on a simulation argument from an execution of a renaming network to an execution of a sorting network. We start from an arbitrary execution $\mathcal{E}$ of the renaming network, and we build a valid execution of a sorting network. The structure of the outputs in the sorting network execution will imply that the tightness and uniqueness properties hold in the renaming network execution.

Let P be the set of processes that have taken at least one step in $\mathcal{E}$. Each process $p_i \in P$ is assigned a unique input port v_i in the renaming network. Let I denote the set of input ports on which there is a process present. We then introduce a new set of "ghost" processes G, each assigned to one of the input ports in $\{1, 2, \ldots, M\} \setminus I$. We denote by C the set of "crashed" processes, that is, processes that took a step in $\mathcal{E}$, but did not return an output port index.

The next step in the transformation is to assign input values to these processes. We assign input value 0 to processes in P (and correspondingly to their input ports), and input value 1 to processes in G.

Note that, in execution $\mathcal{E}$, not all test-and-set objects in the renaming network may have been accessed by processes (e.g., the test-and-set objects corresponding to processes in G), and not all processes have reached an output port (i.e., crashed processes and ghost processes). The next step is to simulate the output of these test-and-set operations by extending the current renaming network execution.

We extend the execution by executing each process in $C \cup G$ until completion. We first execute each process in C, in a fixed arbitrary order, and then execute each process in G, in a fixed arbitrary order. The rules for deciding the result of test-and-set objects for these processes are the following.

—If the current test-and-set T already has a winner in the extension of $\mathcal{E}$, that is, a process that returned 0 and went "up", then the current process automatically goes "down" at this test-and-set.
—Otherwise, if the winner has not yet been decided in the extension of $\mathcal{E}$, then the current process becomes the winner of T and goes "up," that is, takes output port x'.

In this way, we obtain an execution in which M processes participate, and each test-and-set object has a winner and a loser. By Claim 1, the execution is well-formed, that is, there are never two processes (or two values) on the same wire. Also note that the resulting extension of the original execution $\mathcal{E}$ is a valid execution of a renaming network, since we are assuming an asynchronous shared memory model, and the ghost and crashed processes can be seen simply as processes that are delayed until processes in $P \setminus C$ returned.

The key observation is that, for every two-process test-and-set T in the network, T obeys the comparison property of comparators in a sorting network, applied to the values assigned to the participating processes. We take cases on the processes p and q participating in T.

(1) If p and q are both in P, then both have associated value 0, so the T respects the comparison property irrespective of the winner.
(2) If $p \in P$ and $q \in G$, then notice that p necessarily wins T, while q necessarily loses T. This is trivial if $p \in P \setminus C$; if $p \in C$, this property is ensured since we execute all processes in C before processes in G when extending $\mathcal{E}$. Therefore, the process with associated value 0 always wins the test-and-set.
(3) If p and q are both in G, then both have associated value 1, so T respects the comparison property irrespective of the winner.

The final step in this transformation is to replace every test-and-set operation with a comparator between the binary values corresponding to the two processes that participate in the test-and-set. Thus, since we have started from a sorting network, we obtain a sequence of comparator operations ordered in stages, in which each stage contains only comparison operations that may be performed in parallel. This argument shows that all comparators obey the comparison property applied to the values we assigned to the corresponding processes. In particular, when input values are different, the lower value (corresponding to participating processes) always goes "up," while the higher value always goes "down."

Thus, the execution resulting from the last transformation step is in fact a valid execution of the sorting network from which the renaming network has been obtained. Recall that we have associated each process that took a step to a 0 input value, and each ghost process to a 1 input value to the network. Since, by Claim 1, no two input values may be sorted to the same output port, we first obtain that the output port indices that the processes in P return are unique. For namespace tightness, recall that we have obtained an execution of a sorting network with M input values, $M - k$ of which, that is, those corresponding to processes in G, are 1. By the sorting property of the network, it follows that the lower $M - k$ output ports of the sorting network are occupied by 1 values. Therefore, the $M - k$ "ghost" processes that have not taken a step in $\mathcal{E}$ must be associated with the lower $M - k$ output ports of the network in the extended execution. Conversely, processes in P must be associated with an output port between 1 and k in the extension of the original execution $\mathcal{E}$. The final step is to notice that, in $\mathcal{E}$, we have not modified the output port assignment for processes in $P \setminus C$, that is, for the processes that returned a value in the execution $\mathcal{E}$. Therefore, these processes must have returned a value between 1 and k. This concludes the proof of this claim and of the theorem. □

We now apply the renaming network construction starting from sorting networks of optimal logarithmic depth, whose existence is ensured by the AKS construction [Ajtai et al. 1983]. (Recall that the AKS construction [Ajtai et al. 1983] gives, for any integer $N > 0$, a network for sorting N integers, whose depth is $O(\log N)$. The construction is quite complex, and therefore we do not present it here.)

COROLLARY 5.2 (AKS). *The renaming network obtained from an AKS sorting network [Ajtai et al. 1983] with M input ports solves the strong adaptive renaming problem with M initial names, guaranteeing name uniqueness in all executions, and using $O(\log M)$ test-and-set operations per process in the worst case. The termination guarantee is the same as that of the test-and-set objects used.*

PROOF. The fact that this instance of the algorithm solves strong adaptive renaming follows from Theorem 5.1. For the complexity claims, notice that the number of test-and-set objects a process enters is bounded by the depth of the sorting network from which the renaming network has been obtained. In the case of the AKS sorting network with M inputs, the depth is $O(\log M)$. □

5.2. A Strong Adaptive Renaming Algorithm

We present an algorithm for adaptive strong renaming based on an adaptive sorting network construction. For any $k \geq 0$, the algorithm guarantees that k processes obtain unique names from 1 to k. We start by presenting a sorting network construction that adapts its size and complexity to the number of processes executing it. We will then use this network as a basis for an adaptive renaming algorithm

5.2.1. An Adaptive Sorting Network. We present a recursive construction of a sorting network of arbitrary size. We will guarantee that the resulting construction ensures
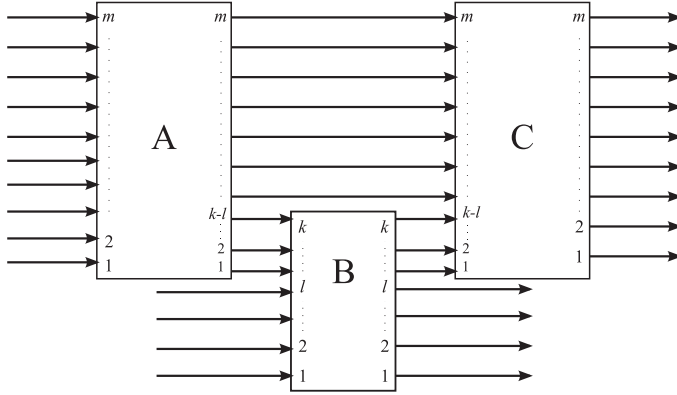
Fig. 8. One stage in the construction of the adaptive sorting network. The small labels indicate port number: upper is higher.

the properties of a sorting network whenever truncated to a finite number of input (and output) ports. The sorting network is adaptive, in the sense that any value entering on wire n and leaving on wire m traverses at most $O(\log \max(n, m))$ comparators.

Let the *width* of a sorting network be the number of input (or output) ports in the network. The basic observation is that we can extend a small sorting network B to a wider range by inserting it between two much larger sorting networks A and C. The resulting network is nonuniform—different paths through the network have different lengths, with the lowest part of the sorting network (in terms of port numbers) having the same depth as B, whereas paths starting at higher port numbers may have higher depth.

Formally, suppose we have sorting networks A, B, and C, where A and C have width m and B has width $k < m$. Label the inputs of A as $A_1, A_2, \ldots, A_m$ and the outputs as $A'_1, A'_2, \ldots, A'_m$, where $i < j$ means that A'_i receives a value less than or equal to A'_j. Similarly, label the inputs and outputs of B and C. Fix $\ell \leq k/2$ and construct a new sorting network ABC with inputs $B_1, B_2, \ldots, B_\ell, A_1, \ldots, A_m$ and outputs $B'_1, B'_2, \ldots, B'_\ell, C'_1, C'_2, \ldots, C'_m$. Internally, insert B between A and C by connecting outputs $A'_1, \ldots, A'_{k-\ell}$ to inputs $B_{\ell+1}, \ldots, B_k$; and outputs $B'_{\ell+1}, \ldots, B'_k$ to inputs $C_1, \ldots, C_{k-\ell}$. The remaining outputs of A are wired directly across to the corresponding inputs of C: outputs $A'_{k-\ell+1}, \ldots, A'_m$ are wired to inputs $C_{k-\ell+1}, \ldots, C_m$. (See Figure 8.)

LEMMA 5.3. *The network ABC constructed as described previously is a sorting network.*

PROOF. The proof uses the well-known Zero-One Principle [Cormen et al. 2009]: we show that the network correctly sorts all input sequence of zeros and ones, and deduce from this fact that it correctly sorts all input sequences.

Given a particular 0-1 input sequence, let z_B and z_A be the number of zeros in the input that are sent to inputs $B_1 \cdots B_\ell$ and $A_1 \cdots A_m$. Because A sorts all of its incoming zeros to its lowest outputs, B gets a total of $z_B + \max(k - \ell, z_A)$ zeros on it inputs, and sorts those zeros to outputs $B'_1 \cdots B'_{z_B+\max(k-\ell,z_A)}$. An additional $z_A - \max(k - \ell, z_A)$ zeros propagate directly from A to C.

We consider two cases, depending on the value of the max.

—*Case 1.* $z_A \leq k - \ell$. Then, B gets $z_B + z_A$ zeros (all of them), sorts them to its lowest outputs, and those that reach outputs $B'_{\ell+1}$ and above are not moved by C. Therefore, the sorting network is correct in this case.

—*Case 2.* $z_A > k - \ell$. Then B gets $z_B + k - \ell$ zeros, while $z_A - (k - \ell)$ zeros are propagated directly from A to C. Because $\ell \leq k/2$, $z_B + k - \ell \geq k/2 \geq \ell$, and B sends ℓ zeros out its direct outputs $B'_1 \cdots B'_\ell$. All remaining zeros are fed into C, which sorts them to the next $z_A + z_B - \ell$ positions. Again, the sorting network is correct. □

When building the adaptive network, it will be useful to constrain which parts of the network particular values traverse. The key tool is given by the following lemma.

LEMMA 5.4. *If a value v is supplied to one of the inputs B_1 through B_ℓ in the network ABC, and is one of the ℓ smallest values supplied on all inputs, then v never leaves B.*

PROOF. Immediate from the construction and Lemma 5.3; v does not enter A initially, and is sorted to one of the output $B'_1 \cdots B'_\ell$, meaning that it also avoids C. □

Now let us show how to recursively construct a large sorting network with polylog M depth when truncated to the first M positions. We assume that we are using a construction of a sorting network that requires at most $a \log^c n$ depth to sort n values, where a and c are constants. For the AKS sorting network [Ajtai et al. 1983], we have $c = 1$ and very large a; for constructible networks (e.g., the bitonic sorting network [Knuth 1998]), we have $c = 2$ and small a.

Start with a sorting network S_0 of width 2. In general, we will let w_j be the width of S_j; so we have $w_0 = 2$. We also write d_j for the depth of S_j (the number of comparators on the longest path through the network).

Given S_j, construct S_{j+1} by appending two sorting networks A_{j+1} and C_{j+1} with width $w_j^2 - w_j/2$, and attach them to the top half of S_j as in Lemma 5.3, setting $\ell = w_j/2$.

Observe that $w_{j+1} = w_j^2$ and $d_{j+1} = 2a \log^c(w_j^2 - w_j/2) + d_j \leq 4a \log^c w_j + d_j$. Solving these recurrences gives $w_j = 2^{2^j}$ and $d_j = \sum_{i=0}^{j} 2^{c(i+2)} a = O(2^{cj})$.

If we set $M = 2^{2^j}$, then $j = \lg \lg M$, and $d_j = O(2^{c \lg \lg M}) = O(\log^c M)$. This gives us polylogarithmic depth for a network with M lines, and a total number of comparators of $O(M \log^c M)$.

We can in fact state a stronger result, relating the input and output port indices for a value with the complexity of sorting that value.

THEOREM 5.5. *For any $j \geq 0$, the network S_j constructed here is a sorting network, with the property that any value that enters on the nth input and leaves on the mth output traverses $O(\log^c \max(n, m))$ comparators.*

PROOF. That S_j is a sorting network follows from induction on j using Lemma 5.3.

For the second property, let $S_{j'}$ be the smallest stage in the construction of S_j to which input n and output m are directly connected. Then $w_{j'-1}/2 < \max(n, m) \leq w_{j'}/2$, which we can rewrite as $2^{2^{j'-1}} < 2\max(n, m) \leq 2^{2^{j'}}$ or $j' - 1 < \lg \lg \max(n, m) \leq j'$, implying $j' = \lceil \lg \lg \max(n, m) \rceil$. By Lemma 5.4, the given value stays in $S_{j'}$, meaning it traverses at most $d_{j'} = O(2^{cj'}) = O(2^{c \lceil \lg \lg \max(n,m) \rceil}) = O(\lg^c \max(n, m))$ comparators. □

5.2.2. Transformation to a Renaming Nework. We now apply the previous results to renaming networks.

COROLLARY 5.6. *Consider the sequence of networks R_j resulting from replacing comparators with two-process test-and-set objects in the extensible sorting network construction from Section 5.2.1. For any $M \geq k > 0$, assuming initial names from 1 to M, these networks solve strong renaming for k processes with $O(\log M)$ test-and-set accesses per process.*

PROOF. Fix a $M \geq k > 0$, and let j be the first index in the sequence such that the resulting network $\bar{S}_j$ has at least M inputs and M outputs. By Theorem 5.5, this network sorts, and has depth $O(\log M)$ (considering the version of the construction using the AKS sorting network as a basis). By Theorem 5.1, the corresponding renaming network R_j solves adaptive strong renaming for any k processes with initial names between 1 and M, performing $O(\log M)$ test-and-set accesses per process. □

5.2.3. An Algorithm for Strong Adaptive Renaming. We show how to apply the adaptive sorting network construction to solve strong adaptive renaming when the size of the initial namespace, M, is unknown, and may be unbounded. This procedure can also be seen as transforming an arbitrary renaming algorithm A, guaranteeing a namespace of size M, into *strong* renaming algorithm $S(A)$, ensuring a namespace from 1 to k. In case the processes have initial names from 1 to M, then A is a trivial algorithm that takes no steps. We first describe this general transformation, and then consider a particular case to obtain a strong adaptive renaming algorithm with logarithmic time complexity. Notice that, in order to work for unbounded contention k, the algorithm may use unbounded space, since the adaptive renaming network construction continues to grow as more and more processes access it.

Description. We assume a renaming algorithm A with complexity $C(A)$, guaranteeing a namespace of size M (which may be a function of k, or n). We assume that processes share an instance of algorithm A and an adaptive renaming network R, obtained using the procedure in Section 5.2.1.

The transformation is composed of two stages. In the first stage, each process p_i executes the algorithm A and obtains a temporary name v_i from 1 to M. In the second stage, each process uses the temporary name v_i as the index of its (unique) input port to the renaming network R. The process then executes the renaming network R starting at the given input port, and returns the index of its output port as its name.

Wait-Freedom. Notice that, technically, this algorithm may not be wait-free if the number of processes k participating in an execution is *infinite*, then it is possible that a process either fails to acquire a temporary name during the first stage, or it continually fails to reach an output port by always losing the test-and-set objects it participates in. Therefore, in the following, we assume that k is finite, and present bounds on step complexity that depend on k.

Constructibility. Recall that we are using the AKS sorting network [Ajtai et al. 1983] of $O(\log M)$ depth for M inputs as the basis for the adaptive renaming network construction. However, the constants hidden in the asymptotic notation for this construction are large, and make the construction impractical [Knuth 1998]. On the other hand, since the construction accepts any sorting network as basis, we can use Batcher's bitonic sorting network [Knuth 1998], with $O(\log^2 M)$ depth as a basis for the construction. Using bitonic networks trades a logarithmic factor in terms of step complexity for ease of implementation.

5.2.4. Analysis of the Strong Adaptive Renaming Algorithm. We now show that the transformation is correct, transforming any renaming algorithm A with namespace M and complexity $C(A)$ into a *strong* renaming algorithm, with complexity cost $C(A) + O(\log M)$.

THEOREM 5.7 (NAMESPACE BOOSTING). *Given any renaming algorithm A ensuring namespace M with expected worst-case step complexity $C(A)$, the renaming network construction yields an algorithm $S(A)$ ensuring* strong *renaming. The number of test-and-set operations that a process performs in the renaming network is $O(\log M)$. Moreover, if A is* adaptive*, then the algorithm $S(A)$ is also adaptive. When using the randomized*

test-and-set construction of Tromp and Vitányi [2002], the number of steps that a process takes in the renaming network is $O(\log M)$ both in expectation and with high probability in k.

PROOF. Fix an algorithm A with namespace M and worst-case step complexity $C(A)$. Therefore, we can assume that, during the current execution, each process enters a unique input port between 1 and M in the adaptive renaming network. By Corollary 5.6, each process reaches a unique output port between 1 and k, which ensures that the transformation solves strong renaming.

If the algorithm A is adaptive, that is, the namespace size M and its complexity $C(A)$ depend only on k, then the entire construction is adaptive, since the adaptive renaming network guarantees a namespace size of k, and complexity $O(\log M)$, which only depends on k. This concludes the proof of correctness.

For the upper bound on worst-case step complexity, notice that a process may take at most $C(A)$ steps while running the first stage of the algorithm. By Corollary 5.6, we obtain that a process performs $O(\log M)$ test-and-set accesses in any execution. Since the randomized test-and-set construction of Tromp and Vitányi [2002], has *constant* expected step complexity, the worst-case expected step complexity of the whole construction is $C(A) + O(\log M)$.

To obtain the high probability bound on the number of read-write operations performed by a process in the renaming network, first recall that the number of test-and-set operations that a process may perform while executing the renaming network is $\Theta(\log M)$. Therefore, we can see the number of read-write steps that a process takes while executing the renaming network as a sum of $\Theta(\log M)$ geometrically distributed random variables, one for each two-process test-and-set. It follows that the number of steps that a process performs while executing the renaming network is $O(\log M)$ with high probability in M. Since $M \geq k$, this bound also holds with high probability in k. □

We now substitute the generic algorithm A with the RatRace loose renaming algorithm of Alistarh et al. [2010], whose structure and properties are given in the Appendix. We obtain a strong renaming algorithm with logarithmic step complexity. First, the properties of the RatRace renaming algorithm are as follows.

PROPOSITION 1 (RatRace RENAMING). *For $c \geq 3$ constant, the RatRace renaming algorithm described previously yields an adaptive renaming algorithm ensuring a namespace of size $O(k^c)$ in $O(\log k)$ steps, both with high probability in k. Every process eventually returns with probability 1.*

This implies the following.

COROLLARY 5.8. *There exists an algorithm T such that, for any finite $k \geq 1$, T solves strong adaptive renaming with worst-case step complexity $O(\log k)$. The upper bound holds in expectation and with high probability in k.*

PROOF. We replace the algorithm A in Theorem 5.7 with RatRace renaming. We obtain a correct adaptive strong renaming algorithm.

For the upper bounds on complexity, by Proposition 1, the RatRace renaming algorithm ensures a namespace of size $O(k^c)$ using $O(\log k)$ steps, with probability at least $1 - 1/k^c$, for some constant $c \geq 3$. The complexity of the resulting strong renaming algorithm is at most the complexity of RatRace renaming plus the complexity of executing the renaming network. By Theorem 5.7, with probability at least $1 - 1/k^c$, this is at most

$$O(\log k) + O(\log k^c) = O(\log k).$$

The expected step complexity upper bound follows identically. Finally, since RatRace is adaptive, the transformation also yields an adaptive renaming algorithm. □

We also obtain the following corollary, which applies to the case when test-and-set is available as a base object.

COROLLARY 5.9. *Given any renaming algorithm A ensuring namespace M with worst-case step complexity $C(A)$, and assuming test-and-set base objects with constant cost, the renaming network construction yields an algorithm $S(A)$ ensuring* strong *renaming with worst-case step complexity $C(A) + O(\log M)$. Moreover, if A is* adaptive*, then the algorithm $S(A)$ is also adaptive.*

5.2.5. A Multi-Shot Algorithm. The strong adaptive algorithm described in the previous section limits each process to performing a single name request per execution. However, we notice that essentially the same algorithm allows processes to perform multiple name requests. If m is the total number of name requests performed during the execution, then the algorithm ensures a namespace of size m, and step complexity $O(\log m)$.

Description. When requesting a name, the process first executes the RatRace renaming algorithm to obtain a unique input port index for the renaming network. It then executes the renaming network to reach a new output port, whose index it returns as its new name. Notice that the process never releases the name it acquired during a previous request, and performs the new request as if it were a new process.

Analysis. The algorithm has the following properties.

COROLLARY 5.10. *The multi-shot algorithm ensures unique names, and termination with probability* 1*. Moreover, for finite $m \geq k$, in any execution with m total requests, all names returned are between* 1 *and m, and the step complexity of the algorithm is $O(\log m)$, both in expectation and with high probability.*

The proof is straightforward once we notice that every execution with m name requests by $k < m$ processes requests is equivalent to an execution with m requests, each by a *distinct* process (since the process's initial identifier is only used for comparisons inside the RatRace renaming object.) The claim then follows from Corollary 5.8, where $k = m$.

6. APPLICATIONS TO COUNTING

6.1. A Bounded Counter

We now build a counter algorithm based on the strong adaptive renaming algorithm in Section 5.2. The algorithm exports read and increment operations, and has a bounded maximum value $v_{\max}$. We note that the counter relaxes the standard linearizability correctness property, and only ensures a weaker property, called *monotone consistency*, which we describe next.

6.1.1. Monotone Consistency. Monotone consistency [Aspnes et al. 2012a] is a correctness condition for concurrent data structures, which is weaker than linearizability. Intuitively, monotone consistency ensures linearizability of the *increment* operations (no updates are lost), but does not ensure that *read* operations can always be linearized. One advantage of this guarantee is that, for some objects, known monotone consistent implementations are more efficient known than their linearizable counterparts [Aspnes et al. 2012a].[7]

───────

[7]However, no complexity separation between these two consistency conditions is known.

For example, a counter data structure (as defined here) is *monotone consistent* if the following hold.

(1) There exists a total order $<$ over all read operations, such that, if some read operation R_1 finishes before another read operation R_2 starts, then $R_1 < R_2$. If $R_1 < R_2$, the value v_1 returned by R_1 is less than or equal the value v_2 returned by R_2.
(2) The value v returned by a read operation R satisfies $x \leq v$, where x is the number of increment operations that finished before the operation R started.
(3) The value v returned by a read operation R satisfies $y \geq v$, where y is the number of increment operations that start before the read operation completes.

6.1.2. Algorithm

Description. The processes share an adaptive renaming object implemented using the construction in the previous section, and a linearizable max-register, implemented using the construction of Aspnes et al. [2012a], whose properties are given in Section 3.4. The max register has maximum value v_{max}, which may be arbitrarily large, but must be given in advance.

For the increment operation, a process acquires a new name from the adaptive renaming object. It then writes the newly obtained name to the max-register and returns. (Notice that, by the argument in Section 5.2.5, this algorithm supports multiple increments by the same process.) For the read operation, the process simply reads the value of the max-register and returns it.

Analysis. The counter object has the following properties.

LEMMA 6.1 (COUNTER PROPERTIES). *The counter implementation is monotone consistent, and has expected step complexity $O(\log v)$ per increment, where v is the number of* increment *operations started before the operation returns. A* read *operation has cost $O(\min(\log v, n))$.*

PROOF. Termination with probability 1 for finite v follows from the properties of the objects we use. For monotone consistency, we need to prove the following:

(1) There exists a total ordering $<$ on the read operations such that if an operation $R1$ finishes before some operation $R2$ starts, then $R1 < R2$, and if $R1 < R2$, then the value returned by $R1$ is less than or equal to the value returned by $R2$. For this, we order the read operations by their linearization points when reading the max-register object. This ordering clearly has the required properties.
(2) The value v returned by a read is always $\geq$ the number of completed increment operations. Let y be the number of completed increment operations. Notice that each completed operation obtains a unique name, and writes it to the max-register (this holds also if a single process performs multiple increment operations). It then follows that the value in the max-register at the time of the read is at least y.
(3) The value v returned by a read is always $\leq$ the number of started increment operations. Let z be the number of started increment operations. Assume for contradiction that a process returns a value v which is larger than z. In this case, there must exist a process that returned a name which is strictly larger than the number of name requests on the adaptive renaming object. This contradicts the *adaptive* property of the object.

Therefore, the counter object is monotone consistent. For the complexity bound on the increment operation, notice that, by Corollary 5.10, the complexity of the first stage of the adaptive renaming protocol is $O(\log v)$, and the number of temporary names is $O(\text{poly } v)$ with high probability. It then follows that the complexity of the adaptive renaming object is $O(\log v)$ in expectation. By Theorem 5.7, the same bound holds

with high probability. By the properties of the max-register, it follows that that the complexity of an increment operation is $O(\log v)$. The complexity of the read operation is the same as the complexity of the max register. □

Linearizability Counterexample. We show a non-linearizable execution of our counter implementation. Consider three processes p_1, p_2, p_3. Processes p_2 and p_3 start the execution concurrently. Process p_2 obtains name 2 and writes it to the max register, while p_3 is still executing the renaming network. After p_2's operation terminates, p_1 starts its increment operation and "steals" name 1 from p_3, and writes this value to the max register (this execution is possible in a renaming network). We insert a read operation R_1 between the end point of p_2's operation and the start point of p_1's operation. This operation must return value 2. We insert a second read operation R_2 between the end point of p_1's operation and before p_3 writes to the max register. The second operation must also return value 2 for the counter. Notice that, in this case, p_1's operation cannot be properly linearized, since it is located between two read operations returning the same value.

6.2. Linearizable Bounded-Value Fetch-and-Increment

We now show how to use an adaptive tight renaming protocol to construct a linearizable m-valued fetch-and-increment object, that is, a fetch-and-increment object that supports only values up to m. The sequential specification of the object is the same as that of fetch-and-increment, except that the object keeps returning $m-1$ once it has reached the threshold value m.

Description. The outline of the construction is as follows. We first use the tight adaptive renaming protocol to build a linearizable ℓ-test-and-set object, which generalizes a standard test-and-set object by providing ℓ winners instead of a single one. We implement such an object by having processes run the adaptive tight renaming algorithm and return *true* if and only if their acquired name is at most ℓ. To ensure this is linearizable, we protect the renaming protocol with a doorway bit, which guarantees that processes arriving after some process returns *false* cannot prevent a process that started the operation earlier from winning.

The second part of the m-valued fetch-and-increment construction is based on a recursive tree construction. For $\ell = m, m/2, m/4$ down to 1, at each stage ℓ of the construction, we are implementing an ℓ-fetch-and-increment object, composed of one $\ell/2$-test-and-set object, and two $\ell/2$-fetch-and-increment objects (the left child, and the right child of the current node, respectively). If a process wins in the $\ell/2$-test-and-set object, then it calls the left $\ell/2$-valued fetch-and-increment object; otherwise, it calls the right object.

Analysis. We begin by formally defining an ℓ-test-and-set object.

Definition 6.2. An *ℓ-test-and-set* object O supports one type of operation which returns either *true* or *false*. The sequential specification of the object is that the first ℓ invocations of the operation return *true* and the rest return *false*.

Our implementation of an ℓ-test-and-set object is given in Figure 9. The following lemma shows correctness of our implementation. Intuitively, any operation that starts late sees the doorway closed, therefore must return *false*.

LEMMA 6.3. *Procedure ℓ-test-and-set in Figure 9 implements a linearizable ℓ-test-and-set.*

PROOF. By correctness of the adaptive tight renaming algorithm, ℓ processes obtain a name whose value is at most ℓ, and therefore exactly ℓ processes return *true*. For

Shared: boolean *doorway*, initially *open*;

procedure ℓ-test-and-set();
if *doorway* = *closed* **then**
    return false
end
else
    name ← strong-renaming();
    if *name* $\leq \ell$ **then** **return** true
    else
       doorway ← *closed*
       return false
    end
end

Fig. 9.   The ℓ-test-and-set implementation.

Shared: *test*, an $\ell/2$-test-and-set object;
O.left, an $\ell/2$-valued f&inc object;
O.right, an $\ell/2$-valued f&inc object;

procedure ℓ-fetch-and-increment();
    if $\ell = 0$ **then** **return** 0;
    if $\ell/2$-*test-and-set(test)* **then**
       return *fetch-and-increment(O.left)*
    end
    else
       return $\ell/2$ + *fetch-and-increment(O.right)*
    end

Fig. 10.   The ℓ-fetch-and-increment object.

linearizability, we partition the operations into two disjoint categories, C_{true} and C_{false}, according to their return values. We order all operations in C_{true} before the time that the doorway is set to *closed*, and all operations in C_{false} afterwards. Within each category we order the operations according to the order of nonoverlapping operations. It is clear that this order satisfies the sequential specification of the ℓ-test-and-set object, since all operations that return *true* are linearized before those that return *false*, and there are exactly ℓ of those. To show that this order preserves the order of nonoverlapping operations, we only need to argue about nonoverlapping operations in different categories, since within each category this order is preserved by construction. Let $idop_1$ be an operation that returns *true* and op_2 be an operation that returns *false* and assume, towards a contradiction, that op_2 finishes before op_1 starts. Then, op_2 must set the doorway to *closed*, implying that after op_1 reads the doorway it returns *false*. This contradiction concludes the proof that this implements a linearizable ℓ-test-and-set object.   □

Next, Figure 10 shows an implementation of an ℓ-valued fetch-and-increment object using two smaller fetch-and-increment objects. Note that this recursive construction unfolds to a tree, whose leaves are 0-valued fetch-and-increment objects. We implement such an object with an empty data structure on which the fetch-and-increment operation always returns 0.

We conclude with a proof of correctness of this implementation. The basic idea is that the linearizability of the $\ell/2$-test-and-set object allows us to linearize all operations incrementing to small values before those that increment to large values.

LEMMA 6.4.   *If O.left and O.right are linearizable $\ell/2$-fetch-and-increment objects, then procedure recursive-fetch-and-increment implements a linearizable ℓ-fetch-and-increment object.*

PROOF.   Since *O.left* and *O.right* are linearizable, we can associate each access to them with its linearization point. We partition the operations into two disjoint categories, C_{left} and C_{right}, according to the $\ell/2$-fetch-and-increment object they access. We linearize operations in C_{left} before those in C_{right}. Within each category, we linearize the operations according to the order of their linearization points with respect to the $\ell/2$-fetch-and-increment object they access (*O.left* for C_{left}, and *O.right* for C_{right}). By correctness of the $\ell/2$-test-and-set object, exactly $\ell/2$ processes return *true* and the rest return *false*.

Hence, this ordering preserves the sequential specification of an ℓ-fetch-and-increment, given the assumption that *O.left* and *O.right* are linearizable $\ell/2$-fetch-and-increment objects. To show this preserves the order of nonoverlapping operations,
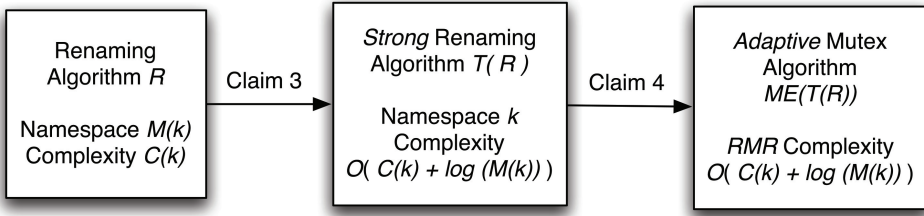
Fig. 11. The structure of the reduction in Theorem 7.1.

we need to argue only about nonoverlapping operations in different categories, since within each category this order is preserved by the assumption on the linearizability of $O.left$ and $O.right$. Let op_1 be an operation in C_{left} and op_2 be an operation in C_{right} and assume, towards a contradiction, that op_2 finishes before op_1 starts. Since op_2 is in C_{right}, then its return value of the $\ell/2$-test-and-set object is *false*. Since op_1 starts after op_2 finishes, it must also return *false* by correctness of the $\ell/2$-test-and-set object, and therefore op_1 must be in C_{right} as well. This contradicts the assumption that op_1 is in C_{left}, which completes the proof. □

Finally, we provide upper bounds for the worst-case time complexity of our implementations. The bounds follow from the properties of the tight adaptive renaming algorithm.

LEMMA 6.5. *The ℓ-test-and-set object has expected step complexity $O(\log k)$. The m-valued fetch-and-increment object has expected step complexity $O(\log k \log m)$, where k is the number of participating processes.*

PROOF. The upper bound on the complexity of ℓ-test-and-set follows from Corollary 5.8. The upper bound on the expected step complexity of the fetch-and-increment object follows from the fact that the recursive construction unfolds to a tree of height $O(\log m)$, with an ℓ-test-and-set at each node, where $\ell \leq m/2$. By this lemma, we obtain that the resulting structure has $O(\log m \log k)$ expected step complexity. □

7. A LOWER BOUND ON THE TIME COMPLEXITY OF DETERMINISTIC RENAMING

In this section, we prove a linear lower bound on the time complexity of deterministic renaming in asynchronous shared memory. The lower bound holds for algorithms using reads, writes, test-and-set, and compare-and-swap operations, and is matched within constants by existing algorithms, as discussed in Section 4. We first prove the lower bound for *adaptive* deterministic renaming, and then extend it to *nonadaptive* renaming by reduction. The lower bound will hold for algorithms that either rename into a subexponential namespace in k (if the algorithm is adaptive) or into a polynomial namespace in n (if the algorithm is not adaptive).

The Strategy. We obtain the result by reduction from a lower bound on mutual exclusion. The argument can be split in two steps, outlined in Figure 11. The first step assumes a wait-free algorithm R, renaming adaptively into a loose namespace of subexponential size $M(k)$, and obtains an algorithm $T(R)$ for *strong* adaptive renaming. As shown in Section 5, the extra complexity cost of this step is an additive factor of $O(\log M(k))$.[8]

---

[8]Since we are assuming a system with atomic test-and-set and compare-and-swap operations, we can use such operations with unit cost in the construction from Section 5.

The second step 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 employ an $\Omega(k)$ lower bound on the RMR complexity of adaptive mutual exclusion by Kim and Anderson [2012]. When plugging in any subexponential 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.

Applications. We notice that we can also apply the result to obtain a linear lower bound on the time complexity of *nonadaptive* renaming algorithms, that guarantee names from 1 to some polynomial function in n, with n known. We prove this generalization by reduction in Section 7.2.

A second application follows from the observation that many common shared-memory objects such as queues, stacks, and fetch-and-increment registers can be used to solve adaptive strong renaming. In turn, this will imply that the linear lower bound will also apply to deterministic shared-memory implementations of these objects using read, write, compare-and-swap or test-and-set operations. We analyze the limitations of this lower bound and ways to circumvent it in Section 7.4.

Finally, the reduction from renaming to mutual exclusion will also imply the existence of a nonadaptive mutual exclusion algorithm with optimal $O(\log n)$ RMR complexity.

7.1. Adaptive Lower Bound

In this section, we prove the following result.

THEOREM 7.1 (INDIVIDUAL TIME LOWER BOUND). *For any $k \geq 1$, given $n = \Omega(k^{2^k})$, 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 $\Omega(k)$ 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, augmented with test-and-set base objects; the complexity cost of the resulting algorithm will be $O(C(k) + \log M(k))$. This result follows immediately from Corollary 5.9.

CLAIM 3. *Assuming test-and-set as a base object, any wait-free algorithm R that renames into a namespace of size $M(k)$ with complexity $C(k)$ can be transformed into a strong adaptive renaming algorithm $T(R)$ with complexity $O(C(k) + \log M(k))$.*

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 increase in terms of step complexity. We note that the mutual exclusion algorithm obtained is *single-use* (i.e., each process executes it exactly once).

CLAIM 4. *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 abuse notation and 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 processes as tickets to enter the critical section.

Processes share a copy of the algorithm R, and a right-infinite array of shared bits $Done[1, 2, \ldots]$, initially false. 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.

We now show that this construction is a correct mutex implementation.

—For the *mutual exclusion* property, let q_i be the process that obtained name i from the renaming network, for $i \in \{1, \ldots, k\}$. Notice that, by the structure of the protocol, for any $i \in \{1, \ldots, k-1\}$, process q_{i+1} may enter the critical section only after process q_i has exited the critical section, since process q_i sets the $Done[i]$ bit to true only after executing the critical section. This creates a natural ordering between processes' accesses in the critical section, which ensures that no two processes may enter it concurrently.

—For the *no deadlock* and *no lockout* properties, first notice that, since the mutex algorithm runs in a failure-free model, and the test-and-set instances we use in the renaming network are deterministically wait-free, it follows that every process will eventually reach an output port in the renaming network. Thus, by Theorem 5.7, each process will eventually be assigned a name from 1 to k. Conversely, each name i from 1 to k will eventually get assigned to a unique process q_i. Therefore, each of the $Done[\,]$ bits corresponding to names $1, \ldots, k$ will be eventually set to true, which implies that eventually each process enters the critical section, as required.

—The *unobstructed exit* condition holds since each process performs a single operation in the exit section.

For the complexity claims, 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 multiplicative 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.

The last remaining claim is that the resulting renaming algorithm is *adaptive*, that is, its complexity only depends on the contention k in the execution, and the algorithm works for any value of the parameter n. This follows since the original algorithm R was adaptive, and by the structure of the transformation. In fact, the transformation does not require an upper bound on n to be known; if such an upper bound is provided, then it can be used to bound the size of the *Done*[ ] array. This concludes the proof of the claim. □

Final Argument. To conclude the proof of Theorem 7.1, 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, in the cache-coherent model, where $2^{f(k)}$ is the size of the namespace guaranteed by the renaming algorithm.

However, the existence of this algorithm contradicts the $\Omega(k)$ lower bound on the RMR complexity of adaptive mutual exclusion by Kim and Anderson [2012, Theorem 2], stated in Theorem 7.2.

THEOREM 7.2 (MUTEX TIME LOWER BOUND [KIM AND ANDERSON 2012]). *For any $k \geq 1$, given $n = \Omega(k^{2^k})$, 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 [Kim and Anderson 2012].*

The algorithm R is adaptive and therefore works for unbounded n. Therefore, the adaptive mutual exclusion algorithm $ME(T(R))$ also works for unbounded n. Hence, the mutual exclusion lower bound described in this article 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 Kim and Anderson [2012], in which each process performs the entry section once.

7.1.1. Technical Notes

Relation between k and n. The lower bound of Kim and Anderson [2012] 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, for a fixed n, the relative value of k for which the linear lower bound is obtained may be very small. For example, the lower bound does not preclude an algorithm with running time $O(\min(k, \log n))$ if n is known in advance.

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, since the renaming network is independent from the original renaming algorithm R.

Single-Use Mutex. As noted previously, the mutual exclusion algorithm we obtained is *single-use*. This is not a problem for the lower bound, since it holds for executions where each process invokes the entry section once; however, it limits the usefulness of the algorithm. We note that the algorithm can be extended to a variant where processes invoke the critical section several times (following the outline in Section 5.2.5); however,

in this case, the time complexity will be logarithmic in the total number of mutual exclusion calls in the execution.

Progress Conditions. Known adaptive renaming algorithms, for example, Moir and Garay [1996] and Alistarh et al. [2011a], do not guarantee wait-freedom in executions where the number of participants is unbounded, since a process may be prevented indefinitely from acquiring a name by new incoming processes. Note that our lower bound applies to these algorithms as well, as the original mutual exclusion lower bound of Kim and Anderson [2012] applies to all mutex algorithms ensuring livelock-freedom, and our transformation does not require a strengthening of this progress condition.

7.2. Nonadaptive Renaming Lower Bound

The same technique also implies a linear lower bound on the step complexity of *nonadaptive* renaming algorithms. For nonadaptive algorithms, the size of the set of participants is bounded by a fixed, known parameter N, and the size of the target namespace and the complexity of the algorithm depend on this parameter.

Strategy. We prove the generalization by reduction: we first show that, under some restrictions, a non-adaptive renaming algorithm can be transformed into a renaming algorithm *with fails*, which returns a fail indication whenever the number of processes k accessing the algorithm is $> N$. Then, we show that any renaming algorithm with fails can be used to obtain an adaptive renaming algorithm with similar step complexity and namespace guarantees. The lower bound then follows from the previous result for adaptive algorithms.

7.2.1. Renaming with Fails. We define a renaming algorithm *with fails* as a nonadaptive renaming algorithm $RF(N)$, 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 that the algorithm allows. On the other hand, if $k > N$, then the algorithm $RF(N)$ may return a special value fail to the calling process instead of a (unique) name. An *instance* of the algorithm is a variant $RF(N)$ for a particular N.

Definition 7.3 (*Renaming with Fails*). The *renaming with fails* task for parameter N in namespace $T(N)$ assumes $k > 0$ processes with unique initial identifiers from an unbounded namespace, and ensures the following:

(1) *Termination*. In every execution, every correct process returns either an integer name or a fail indication.
(2) *Namespace Size*. In every execution, every integer name returned is from 1 to $T(N)$.
(3) *Uniqueness*. In every execution, no two processes may return the same integer name.
(4) *Progress when $k \leq N$*. If the contention k in the current execution is at most N, no process returns fail.

Note that, by the specification of the renaming problem, an instance assumes no limit on the size of the initial identifiers that participating processes may have; however, a non-fail return value is guaranteed only if at most N processes participate. Also, this property ensures that the complexity of a renaming algorithm may not depend on the size of the initial namespace that the algorithm accepts. Otherwise, since the algorithm accepts a virtually infinite namespace, the algorithm would have unbounded complexity.

From nonadaptive renaming to renaming with fails.. In the following, we consider nonadaptive renaming algorithms that ensure the following properties:

(1) every step of the algorithm executed by a correct process terminates eventually;
(2) any step can only access variables that are allocated by the algorithm;
(3) there exists a bounded function $C(N)$ such that, for every execution, the number of steps performed by a process is at most $C(N)$.

It is straightforward to see that any wait-free algorithm can be modified to ensure properties (1) and (2). Property (3) states that the step complexity of the algorithm can be bounded by a function $C(N)$. Given these properties, we give a procedure to transform any nonadaptive renaming algorithm R into a renaming algorithm with fails RF with the same asymptotic complexity.

Let $T(N)$ be the namespace that R renames into; $C(N)$ is an upper bound on 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. Processes share an instance of the algorithm R, and an array $Split$ of $T(N)$ randomized splitter objects, as defined in Section A.1 in Appendix A.

Transformation. Each process executes algorithm R, checking the local program counter at every step. If the local program counter exceeds the value $C(N)$ while running R, then the process automatically returns fail. If the process obtains a name that is larger than $T(N)$ from R, then it automatically returns fail. If the process obtains a name r from the current instance of R, then it checks that this name is unique by accessing the auxiliary array $Split$ of randomized splitter objects in position r. If the splitter returns stop, then the process returns that name as its decision value (recall that the splitter properties ensure that at most one process may return stop at a splitter). Otherwise, if the splitter returns left, or right, then the process returns fail.

We now check that this transformation results in a renaming algorithm with fails, whose asymptotic step complexity is the same as the one of the original algorithm R. Notice that, in general, the behavior of a nonadaptive renaming algorithm is not specified when k exceeds N.

LEMMA 7.4 (RENAMING WITH FAILS). *For any $N > 0$, given a renaming algorithm R for at most N processes with complexity $C(N)$ ensuring a namespace of size $T(N)$, the transformation yields a renaming with fails algorithm RF with parameter N, having the same asymptotic step complexity as R, ensuring the same properties as R as long as $k \leq N$.*

PROOF. Consider a renaming algorithm R as described previously. If $k \leq N$, it is easy to see that no process returns a fail indication, and the algorithm RF ensures the same properties as R, with the same asymptotic complexity.

Otherwise, if $k > N$, then the algorithm R may break correctness either by returning a name that is outside the range $T(N)$, or by having two processes return the same integer name, or by having a process run forever. Other deviations from correctness are excluded, since we assume that every step by a correct process eventually returns, and steps may only access memory allocated by the algorithm. However, we cover these possibilities in the transformation by having processes return fail indications whenever one of these events occurs: a process returning a name out of range returns fail; processes getting the same name detect the conflict through the $Split$ array; a process returns fail if it takes more than $C(N)$ steps as part of the algorithm R. Therefore, the transformation implements a renaming algorithm with fails for parameter N. □

7.2.2. From Renaming with Fails to Adaptive Renaming. In this section, we show that any renaming algorithm with fails ensuring a namespace of polynomial size in N can be transformed into an *adaptive* renaming algorithm, at the cost of a multiplicative logarithmic increase in running time, conserving polynomial namespace size.

Transformation. We start from a nonadaptive renaming with fails algorithm R, which, for any $N \geq 1$, renames into a namespace of size $T(N)$, with complexity $C(N)$, as long as the number of participants k to the instance does not exceed N. We consider an infinite series $(R_i)_{i \in \mathbb{N}}$ 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 $(R_i)_{i \in \mathbb{N}}$ in order, until it first obtains an integer name from an instance R_i (as opposed to a fail indication). If, on the other hand, a process obtains a fail 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, that is, for $j \geq 2$, $\sum_{j=1}^{i-1} T(2^j)$.

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

LEMMA 7.5. *Let A be an algorithm that renames with fails such that, for any $N \geq 1$, it guarantees a namespace of size polynomial in N with step complexity $o(N)$. Then, this 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 the highest index of an instance in the series $(R_i)_{i \in \mathbb{N}}$ that a process accesses in this execution. Since a process may only access an instance of a higher index if it fails in the current instance, and an instance R_i may return fail only if the number of participants k exceeds the number of allowed participants 2^i, it follows that $m = O(\log k)$.

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 fails guarantees uniqueness, and the namespace resulting from the transformation is partitioned into the namespaces returned by the instances $(R_i)_{i \in \mathbb{N}}$.

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 $\sum_{i=1}^{m} 2^{ci} \leq 2^{c(m+1)} = O(k^c)$, that is, polynomial in k, since $m = O(\log k)$. Notice that the transformation uses no knowledge of the maximum number of processes that may participate in the execution. Therefore, the transformation is an *adaptive* renaming algorithm that renames into a namespace of size 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 $C(2^m) + C(2^{m-1}) + \cdots + C(1)$. Since R_m is the highest accessed instance of algorithm R, it follows that $2^{c(m+1)} \geq k \geq 2^{m-1}$. (The first inequality follows since k processes accessing the first m objects can occupy at most $2^{c(m+1)}$ distinct names. The second holds since object R_{m-1} is not sufficient for the processes.) We want to show that $C(2^m) + C(2^{m-1}) + \cdots + C(1)$ is in $o(2^m) = o(k)$, knowing that $C(2^m) = o(2^m)$. Therefore, we need to show that the quantity

$$\frac{C(2^m) + C(2^{m-1}) + \cdots + C(1)}{2^m}$$

converges to 0 as $m \to \infty$. Let $A_m = C(2^m) + C(2^{m-1}) + \cdots + C(1)$, and let $B_m = 2^{m+1} - 1$. Recall the following result from basic calculus.

LEMMA 7.6 (STOLZ-CESÀRO). *Let $(A_n)_{n\geq 1}$ and $(B_n)_{n\geq 1}$ be two sequences of real numbers, such that $(B_n)_{n\geq 1}$ is positive, strictly increasing, and unbounded. Then*

$$\lim_{n\to\infty}\frac{A_n}{B_n}=\lim_{n\to\infty}\frac{A_{n+1}-A_n}{B_{n+1}-B_n},$$

if the limit on the right-hand side exists.

We apply this result to A_m and B_m as defined previously, and obtain that

$$\lim_{m\to\infty}\frac{C(2^m)+C(2^{m-1})+\cdots+C(1)}{2^{m+1}-1}=\lim_{n\to\infty}\frac{A_{m+1}-A_m}{B_{m+1}-B_m}=\lim_{m\to\infty}\frac{C(2^{m+1})}{2^{m+1}}=0,$$

where the second limit exists and is 0, since $C(k)=o(k)$. Therefore, by a change of variables, the complexity of the transformation is $o(k)$, as claimed. □

On the other hand, the existence of such an adaptive renaming algorithm contradicts Theorem 7.1. Therefore, it follows that every deterministic renaming algorithm with fails with parameter N, guaranteeing a namespace polynomial in N has complexity $\Omega(N)$. From Lemma 7.4, the same result holds for nonadaptive renaming algorithms which ensure properties (1)–(3).

COROLLARY 7.7. *Any deterministic nonadaptive renaming algorithm, with the property that for any $n\geq 1$ the algorithm ensures a namespace polynomial in n, has worst-case step complexity $\Omega(n)$.*

7.3. Applications

7.3.1. Lower Bounds for Other Objects. These results imply time 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.

LEMMA 7.8. *For any $k>0$, we can solve adaptive strong renaming using a fetch-and-increment register, a queue, or a counter.*

PROOF. Given a linearizable fetch-and-increment register, we can solve adaptive strong renaming by having each participant call the fetch-and-increment operation once, and return the value received plus 1. The renaming properties are follow trivially from the sequential specification of fetch-and-increment.

Given a linearizable shared queue, we can solve renaming as follows. If an upper bound on n is given, then we initialize the queue with distinct integers $1,2,\ldots,n$; otherwise, we initialize it with an unbounded string of integers $1,2,3,\ldots$. In both cases, 1 is the element at the head of the queue. Given this initialized object, we can solve adaptive strong renaming by having each participant call the dequeue operation once, and return the value received. Correctness follows trivially from the sequential specification of the queue.

Finally, given a stack, we initialize it with the same string of integers, where 1 is the top of the stack. To solve renaming, each process performs pop on the stack and returns the element received. □

This implies a linear-time lower bound for these objects.

COROLLARY 7.9 (QUEUES, STACKS, FETCH-AND-INCREMENT). *Consider a wait-free linearizable implementation A of a fetch-and-increment register, queue, or stack, in shared*

Shared:;
An AKS renaming network R, with n input and output ports;
An array $Done$ of n registers, initially false;

procedure entry-section(v_i);
    $w \leftarrow$ input wire corresponding to v_i;
    while *w is not an output wire* **do**
        $T \leftarrow$ next test-and-set on wire w;
        $res \leftarrow T.$test-and-set( );
        if $res = 0$ **then**
            $w \leftarrow$ output wire x' of T;
        end
        else
            $w \leftarrow$ output wire y' of T;
        end
    end
    `/* w is an output wire                                                                   */`
    $id_i \leftarrow w.index$;
    if $id_i = 1$ **then**
        execute *critical section*;
    end
    else
        spin until $Done[id_i - 1] =$ true;
        execute *critical section*;
    end
    return;

procedure exit-section( );
    $Done[id_i] \leftarrow$ true;
    return;

Fig. 12.   Pseudocode for the mutex algorithm.

memory with read, write, test-and-set, and compare-and-swap operations. If the algorithm A is deterministic, then, for any $k \geq 1$, given $n = \Omega(k^{2^k})$, there exists an execution of A with $2k - 1$ participants in which (1) each participant performs a single call to the object, and (2) some process performs k RMRs (or steps).

7.3.2. A Time-Optimal One-Shot Nonadaptive Mutex Algorithm. Another application of the lower bound argument is that we can obtain an asymptotically optimal one-shot mutual exclusion algorithm from an AKS sorting network [Ajtai et al. 1983]. We present this algorithm in the cache-coherent (CC) model.

Description. 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 [Golab et al. 2007]. In the mutual exclusion problem, processes are assumed hold unique initial identifiers v_i from 1 to n; therefore, we use these initial identifiers to assign unique input ports to processes. (Please see Figure 12 for the pseudocode.) A process progresses through the network as described in Section 5.1.1. 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 Claim 4, given in Section 7.1. In particular, the asymptotic local RMR complexity of this algorithm is the same as the depth of the AKS sorting network (plus at most two RMRs for reading the *Done* bits), that is, $O(\log n)$, therefore the algorithm is optimal by the lower bound of Attiya et al. [2008]. Yang and Anderson [1994] presented an upper bound with the same asymptotic complexity, but significantly better constants, using a different technique. The same construction can be used starting from constuctible sorting networks, for example, bitonic sorting networks [Knuth 1998], at the cost of increasing complexity

by a logarithmic factor. Notice that the linear RMR lower bound of Kim and Anderson [2012] does not yield a $\Omega(n)$ RMR lower bound for *nonadaptive* mutual exclusion (which would contradict the existence of our algorithm).

7.4. Circumventing the Lower Bound

Intuitively, the lower bound shows that, if n processes could potentially participate, then, for k such that $n \geq k^{2^k}$, one can obtain a schedule with k participating processes in which one process takes a step for roughly *every other* participating process. There are (at least) two ways for algorithms to circumvent the lower bound.

Randomization. As shown in Section 5, one can avoid the worst-case linear schedules with high probability by allowing processes to flip coins as part of the execution. In particular, for renaming, the complexity of an operation is $O(\log k)$ with high probability, that is, exponentially less than the worst-case deterministic schedule.

Limiting the Parameter n or the Size of the Initial Namespace. Another way of circumventing the lower bound is by assuming that processes already have unique names from 1 to n. (For example, this can be achieved by running a renaming algorithm at the beginning of the execution.) An example of such an algorithm is the $O(\log n \log v)$ counter algorithm by Aspnes et al. [2012a], where n is the number of processes, and v is the maximum value of the counter. This algorithm assigns a unique "input port" in the data structure to each of the n processes. For fixed n, if $k = \Theta(\log n)$ processes are participating, each of them performs a number of operations which is at least linear in k, although this number is in $O(\log n \log v)$. Notice that, if n is small, linear contention in k could be seen as negligible.

8. A TIME LOWER BOUND FOR ADAPTIVE RANDOMIZED RENAMING

In this section, we present lower bounds on the expected *total* step complexity of randomized renaming and counting. (Naturally, the result also applies to deterministic algorithms, yielding a worst-case total step complexity lower bound.) The lower bound holds for algorithms using reads, writes, test-and-set, or compare-and-swap operations, and is matched by the renaming network algorithm.

Strategy. We analyze an adversarial strategy that schedules the processes 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 (i.e., the variant where processes may return names between 1 and ck, where c is a constant). We then notice that a similar argument can be applied to obtain a lower bound for approximate counting (defined here).

Our argument generalizes a previous result by Jayanti [1998], which in turn is similar to a lower bound by Cook et al. [1986] on the complexity of computing basic logical operations on PRAM machines. Jayanti proved an $\Omega(\log k)$ lower bound on the expected step complexity of shared counters, queues, and stacks, which can be tweaked to apply to strong adaptive renaming. We generalize his result in two ways: first, we consider total step complexity, and thus obtain a stronger $\Omega(\log k)$ lower bound on the *average* worst-case expected step complexity of the problem. Second, our results also apply to loose (approximate) versions of renaming and counting, showing that the time complexity benefits of relaxing the object semantics in this way are at most constant.

We begin by discussing some basic definitions and properties related to adversarial schedules in Section 8.1. We then describe the adversarial scheduler and analyze its

properties in Section 8.2. From these properties, we obtain the lower bound for adaptive renaming in Section 8.3. We refine this argument to obtain a lower bound for approximate counting in Section 8.4. We then discuss applications to more complex objects in Section 8.6.

8.1. Preliminaries

8.1.1. Loose Adaptive Renaming and Approximate Counting. We now define *loose* adaptive renaming and *approximate* counting. For $c \geq 1$, the c-loose adaptive renaming problem requires processes to return unique names from 1 to ck, where k is the contention in the execution.

Let C be a counter implementation, supporting operations read and increment. The counter is c-approximate if, for any read operation R, its return value v satisfies the relation

$$\gamma/c \leq v \leq c\gamma,$$

where γ is the number of increment operations linearized before the read operation R.

8.1.2. Operations and Schedules. Recall that we consider a shared-memory model with atomic read, write, and compare-and-swap operations (the test-and-set operation can be trivially replaced by compare-and-swap). Notice that the read and write operations always succeed, whereas the compare-and-swap operation is *conditional*, meaning that it may or may not change the value of the register on which it is called, depending on the value of the register when the operation is called. Notice that certain operations change the value of the underlying object (such as a write with a new value), whereas others are "invisible", such as reads, and failed compare-and-swap operations. In the following, we make this intuition precise, and analyze the existence of schedules in which few operations are visible. We follow the presentation of Attiya and Hendler [2010], where these notions were first defined.

Definition 8.1 (*Invisible Operations [Attiya and Hendler 2010]*). Let e be an operation applied by a process p to an object O, in an execution $E = E_1 e E_2$. We say that e is *invisible* in e if either the value of the object O is not changed by e, or if $E_2 = E' e' E''$, where e' is a write operation on O, and the prefix E' contains no operations by p, and no operation in the prefix E' is applied to O. If e is not invisible in E, we say that it is *visible* in E.

Based on this definition, we now define a *weakly visible* schedule, which minimizes the number of operations that a process sees during an execution.

Definition 8.2 (*Weakly Visible Schedule [Attiya and Hendler 2010]*). Let $S = \{e_1, \ldots, e_\ell\}$ be a set of operations by different processes that are enabled after some execution prefix E, all about to apply write or compare-and-swap operations. We say that an ordering of the operations in S is a *weakly visible* schedule of S after E, denoted by $\sigma(E, S)$, if the following holds. Let $E_1 = E\sigma(E, S)$.

—At most a single operation of S is visible on any one object in E_1. If $e_j \in S$ is visible on a base object in E_1, then e_j is issued by a process that is not aware of any event of S in E_1.
—Any process is aware of at most a single event of S in E_1.

Given these definitions, Attiya and Hendler [2010] proved the following result on the existence of weakly visible schedules. The proof follows by constructing a suitable ordering of the operations in S.

LEMMA 8.3 (WEAKLY VISIBLE SCHEDULES [ATTIYA AND HENDLER 2010]). *Let $S = \{e_1, \ldots, e_\ell\}$ be a set of operations by different processes that are enabled after some execution E,*

```
procedure adversarial-scheduler();
r ← 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;
    end
    R ← processes that have read operations enabled;
    W ← processes 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 ∪ C to perform their operations, in the order defined by the weakly-visible schedule σᵣ ;
    r ← r + 1;
end
```

Fig. 13.   The adversarial strategy for the global lower bound.

all about to apply write or compare-and-swap operations. Then, there is a weakly visible schedule of S after E.

8.1.3. Worst-Case Expected Step Complexity. In the following lower bounds, we will use the following basic fact, whose proof follows from the definition of the expectation of a random variable.

PROPOSITION 2 (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 $S(A)$ such that, in every execution under $S(A)$ in which every process terminates, the processes take at least γ steps, then the (worst-case) expected step complexity of A is at least $\alpha\gamma$.*

8.2. The Adversarial Scheduler

We consider an algorithm A in shared-memory allowing atomic read, write, and compare-and-swap operations. The adaptive adversary follows the pseudocode described in Figure 13. The adversary schedules the processes in rounds: in each round, each process that has not yet returned from A is scheduled to perform the next shared-memory operation that they have enabled. 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 7–7).

In each round, the adversary partitions processes that have an operation enabled into three sets: R, the *readers*, W, the *writers*, and C, the *swappers*. Processes in R are scheduled by the adversary to perform their enabled read operations, in the order of their initial identifiers (line 7). Then, the adversary also schedules the updating processes (writers and swappers) in the order given by a weakly visible schedule σ_r for these operations in the round. (This schedule, which minimizes information flow between processes, was defined and shown to exist in Lemma 8.3.) Once every process has either been scheduled or has returned, the adversary moves on to the next round.

Before we proceed with the analysis, we explain the role of the weakly visible schedule for the processes performing write and compare-and-swap operations in round r. For example, 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, by reading a value that these processes successively modified. 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 *one* other operation, using a *weakly visible* schedule, as described in Section 8.1.2.

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 $\mathcal{S}(A)$. More precisely, 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 Jayanti [1998], 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, that is, $UP(p, r) = UP(p, r - 1)$.
(3) If process p performs a compare-and-swap operation which changes the value of register R in round r, then the information contained in the register is merged with p's information, that is $UP(R, r) = UP(p, r - 1) \cup UP(R, r)$. We also assume that the process p gets the information previously contained in the register $UP(p, r) = UP(p, r - 1) \cup UP(R, r)$ (note that the contents of $UP(R, r)$ might have been already updated in round r).
(4) If process p performs a compare-and-swap operation that does not change the value of 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 compute an upper bound on the size of the *UP* sets for processes and registers, as the rounds progress.

LEMMA 8.4 (BOUNDING INFORMATION). *Given a run of the algorithm A controlled by the adversarial scheduler in Figure 13, for any round $r \geq 0$, and for every process or shared register X, $|UP(X, r)| \leq 4^r$, where the set UP is considered at the beginning of the round r.*

PROOF. The proof follows by induction on the round number $r \geq 0$. For $r = 0$, the claim holds by definition.

Given $r > 0$ for which the claim holds, we prove it for $r + 1$. We first prove the claim for $UP(R, r + 1)$, where R is a register. Obviously, read operations do not add information to the register. We therefore focus on update operations, that is, writes and compare-and-swaps. By Lemma 8.3, the schedule σ_r ensures that at most a single update operation in this round is visible on the register. Therefore, by the update rules,

the size of $UP(R, r + 1)$ can be at most $|UP(R, r)| + |UP(p, r)|$, for some process p. This is at most $2^{r+1} < 4^{r+1}$, as claimed.

We now consider $UP(p, r + 1)$ for a process p. If the process performs a read on a register R in the first part of the phase, then the size of $UP(p, r + 1)$ is at most $|UP(p, r)| + |UP(R, r)| \leq 2^{r+1} < 4^{r+1}$, as claimed.

If the process performs a write or a compare-and-swap on a register R, then there are two cases. If the process performs an event that is visible on the register, then, by Lemma 8.3, the process is not aware of any update on the register R in this round. This implies that the size of the set $UP(p, r + 1)$ is at most $|UP(p, r)| + |UP(R, r)| < 4^{r+1}$, as claimed.

If the process does not perform a visible event on the register, the process may see some other process's update on the register in this round. By Lemma 8.3, the process is aware of at most a single update on the register in this round. Assume this update is performed by some process q. Therefore, $|UP(p, r + 1)| \leq |UP(q, r)| + |UP(p, r)| + |UP(R, r)| \leq 3 \cdot 4^r < 4^{r+1}$. We have covered all cases, and hence the claim follows. □

Indistinguishability. Let $\mathcal{E}$ be an execution of the algorithm obtained using this adversarial strategy, that is, $\mathcal{E} \in \mathcal{S}(A)$. 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 identical to the one presented by Jayanti [1998] in the context of local lower bounds for shared-memory with LL/SC operations. Therefore, we only give an overview of the construction in this article.

LEMMA 8.5 (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.*

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. The construction is an adaptation to the read-write model of the one presented by Jayanti [1998]. 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 Figure 14. 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

---

Parameters: the execution $\mathcal{E}$, the set S;
procedure build($\mathcal{E}, S$);
  $r \leftarrow 1$;
  while true **do**
     // we schedule processes that have not seen a process outside of S in the first
       $r-1$ rounds of $\mathcal{E}$
     $S_r \leftarrow \{\text{ processes } q \mid UP(q, r-1) \subseteq S \}$;
     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($\mathcal{E}, q, j$);
     end
     $\mathcal{R} \leftarrow$ processes in S_r that have read operations enabled;
     $\mathcal{W} \leftarrow$ processes in S_r 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 $\mathcal{W} \cup \mathcal{C}$ to perform their operations, in the order defined by the *weakly-visible* schedule σ_r;
     $r \leftarrow r + 1$;
  end

Fig. 14. The procedure for building the indistinguishable execution.

shared-memory operation, the processes that have not yet terminated are then split into a set of readers, a set of writers, and a set of swappers, that is processes having compare-and-swap operations enabled. The readers are then scheduled in the order of their initial identifiers, after which the writers and the swappers are scheduled in the order of their weakly visible schedule for that round. 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 identical to the one outlined in Jayanti [1998, Lemma 5.2]. More precisely, the execution $\mathcal{E}$ is the $(All, \mathcal{A})$ run, and the execution $\mathcal{E}'$ is the $(S, \mathcal{A})$-run. We refer the reader to Jayanti [1998] for the proof. □

8.3. 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 8.6 (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, that is, 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 $\mathcal{E}$. Without loss of generality, assume that the names returned by processes $q_1, \ldots, q_k$ are in monotonically increasing order; since the names 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 $S = UP(q_i, \ell_i)$, as defined in Section 8.2. Since $\mathcal{E} \in \mathcal{S}(A)$, by Lemma 8.4, 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 8.5, 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 may 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 $\log_4(m_i/c) = (1/2)\log(m_i/c)$. Therefore, for any $i \in 1, \ldots, k$, process q_i returning name m_i has to perform at least $(1/2)\log(m_i/c)$ shared memory operations. Then, the total number of steps that the k processes perform in execution $\mathcal{E}$ is

$$\sum_{i=1}^{k} \ell_i \geq \left(\frac{1}{2}\right) \sum_{i=1}^{k} \log\left(\frac{i}{c}\right) = \Omega\left(k \log\left(\frac{k}{c}\right)\right),$$

where, in the last step, we have used the standard Stirling approximation of $k!$. Since this complexity lower bound holds for every execution resulting from the adversarial strategy, using Proposition 2, we obtain that the expected total step complexity of the algorithm A is $\Omega(\alpha k \log(k/c))$. $\square$

8.4. 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.

One key difference from the proof in the previous section, 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 8.7 (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 the adversary applies the strategy described in Figure 13, that is, $\mathcal{E} \in \mathcal{S}(A)$.

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 execution $\mathcal{E}$. 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 $S = UP(p_i, \ell_i)$, as defined in Section 8.2. By Lemma 8.4, we have that $|S| \leq 4^{\ell_i}$.

Assume for the sake of contradiction that $|S| < v_i/c$. By Lemma 8.5, 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 $\ell_i \geq \log_4(v_i/c) \geq (1/2)\log(i/c^2)$, for every $1 \leq i \leq k$. We obtain that the total number of steps is bounded as follows.

$$\sum_{i=1}^{k} \ell_i \geq \left(\frac{1}{2}\right) \sum_{i=1}^{k} \log\left(\frac{i}{c^2}\right) = \Omega\left(k \log\left(\frac{k}{c^2}\right)\right). \quad \square$$

8.5. Circumventing the Lower Bound

In brief, the lower bound in this chapter states that for implementations of adaptive renaming and related counting objects, the average worst-case expected step complexity is *logarithmic* if the adversary is adaptive, that is, may adapt its schedule based on the results of the processes' coin flips. Thus, this logarithmic threshold can be seen as stronger than the linear deterministic one, since it cannot be avoided by the use of randomization in the presence of a strong adversary.

On the other hand, if the adversary is weak, or *oblivious*, and may not adapt the schedule based on the results of the processes coin flips (i.e., fixes the schedule before the execution), then there exist object implementations that avoid this lower bound. In particular, the approximate counter of Bender and Gilbert [2011] guarantees a constant approximation factor with expected running time $O(\log \log n)$ against an oblivious adversary.

Another way of potentially circumventing the lower bound would be to allow the algorithm to break the approximation guarantee with small probability. Notice that the lower-bound argument, as written, only applies to algorithms that guarantee approximation within a factor of c in all executions. The counter of Bender and Gilbert [2011] is an example of such an object, since it guarantees the constant approximation only with high probability. Another example is the AdaptiveSearch renaming algorithm of Alistarh et al. [2010], which only ensures a namespace from 1 to ck with high probability in k.

8.6. Applications to Other Objects

Given that adaptive renaming can easily be solved using queues, stacks, or fetch-and-increment objects, as shown in Section 7.3, the lower bound in Section 8.3 applies to these objects.

COROLLARY 8.8 (APPLICATIONS). *Consider a wait-free linearizable (randomized) implementation A of a fetch-and-increment register, queue, or stack, in shared memory with read, write, test-and-set and compare-and-swap operations. Then, for any $k \geq 1$, 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.*

9. CONCLUSION

We have given the first tight bounds on the complexity of the renaming problem in asynchronous shared-memory. In particular, we showed that deterministic

implementations of renaming have linear-time complexity as long as they ensure a polynomial-size namespace. Using randomization, we achieve a tight namespace in logarithmic expected time, which is optimal.

Several open questions remain. First, our deterministic lower bound is matched by several algorithms in the literature. For algorithms using only reads and writes, which have been studied more extensively, the algorithm of Chlebus and Kowalski [2008] matches the time lower bound, giving a namespace of size $(8k − \log k − 1)$; an elegant algorithm by Attiya and Fouren [2001] achieves a tighter namespace of size $(6k − 1)$; however, this last algorithm only matches the time lower bound within a logarithmic factor. The fastest known algorithm to achieve an optimal namespace using only reads and writes (of size $(2k−1)$) was given by Afek and Merritt [1999], with time complexity $O(k^2)$. Thus, obtaining a read-write deterministic algorithm which is optimal both in terms of time complexity and namespace size is an intriguing open question.

We also presented a randomized renaming algorithm with logarithmic complexity with high probability, which achieves a tight adaptive namespace. The total time complexity lower bound in Section 8 shows that this algorithm is optimal against an adaptive adversary, and that no asymptotic time complexity gains may be obtained for adaptive renaming by relaxing the namespace size within constant factors. This implies that our randomized solution is optimal from two points of view: time complexity and namespace size.

One disadvantage of the renaming network algorithm is that it is based on an AKS sorting network [Ajtai et al. 1983], which has prohibitively high constants hidden inside the asymptotic notation [Knuth 1998]. Thus, it would be interesting to see whether one can obtain constructible randomized solutions that are time-optimal and namespace-optimal. On the other hand, the total lower bound holds only for *adaptive* algorithms; it is not known whether faster nonadaptive algorithms exist, which could in theory go below the logarithmic threshold. We conjecture that $\Omega(\log n)$ steps is a lower bound for nonadaptive randomized algorithms as well.

The global lower bound applies to deterministic adaptive algorithms as well, implying an $\Omega(k \log k)$ worst-case global time complexity lower bound. On the other hand, the fastest known algorithm to rename into a namespace of size ck for $c \geq 1$ constant has $O(k^2)$ total step complexity [Chlebus and Kowalski 2008]. Closing the gap between upper and lower bounds is still an open question.

Another open question concerns randomized shared-memory renaming in weaker adversarial models, in particular in the oblivious adversary model, that is, when the scheduler has knowledge of the algorithm but fixes the schedule without knowing the results of random coin flips. In this case, there are indications that the logarithmic threshold could be broken, since sub-logarithmic algorithms have been shown to exist for test-and-set [Alistarh and Aspnes 2011] and approximate counters [Bender and Gilbert 2011] against the oblivious adversary.

One aspect of these concurrent data structures, which has been somewhat neglected by research is space complexity, that is, the number of registers necessary for correct shared-memory implementations. Our renaming network algorithm uses $O(k \log k)$ registers assuming hardware test-and-set operations. The linear space complexity lower bounds of Jayanti et al. [2000] apply to adaptive renaming, therefore the renaming network is space-optimal within a logarithmic factor. Tight space complexity thresholds for renaming remain an open question.

Our lower bounds apply to implementations of more complex objects, such as queues, stacks, or fetch-and-increment counters. The total step lower bound suggests that there are complexity thresholds which cannot be avoided even with the use of randomization. In particular, the average step complexity for adaptive versions of these data structures is *logarithmic*, even when using randomization. However, for many such
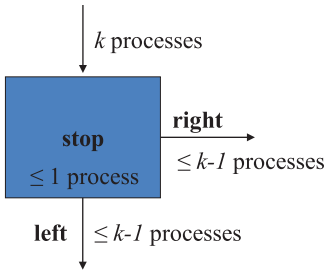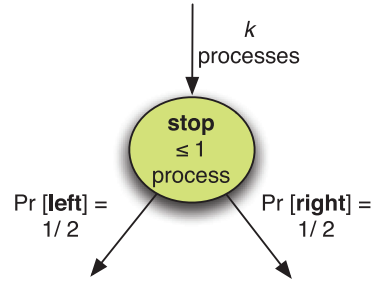
Fig. 15.   Deterministic splitter.

Fig. 16.   Randomized splitter.

objects, there do not exist algorithms that match this logarithmic lower bound. In terms of circumventing this bound, recent results [Alistarh and Aspnes 2011; Bender and Gilbert 2011] suggest that weaker adversarial models and relaxing object semantics, for example, allowing approximate implementations, could be used to go below this logarithmic threshold.

APPENDIXES

A. RANDOMIZED LOOSE RENAMING IN LOGARITHMIC EXPECTED TIME

In this section, we present a randomized algorithm that renames into a namespace of size polynomial in k. The algorithm has logarithmic step complexity in expectation. It is a simple adaptation of the known RatRace test-and-set algorithm of Alistarh et al. [2010] to solve the renaming task. We present the structure of the algorithm; the proof of correctness follows from the original analysis [Alistarh et al. 2010]. We present an abridged version of the proof, for completeness.

A.1. The Randomized Splitter

The *randomized splitter* object is a weak synchronization primitive which allows a process to *acquire* it if running alone, which splits the participants probabilistically if more than one process accesses the object. More precisely, a randomized splitter has the following properties.

—At most one process returns stop.
—If a single correct process calls split, then the process returns stop.
—If a correct process does not return stop, then the probability that it returns left equals the probability that it returns right, which equals $1/2$.

The randomized splitter was introduced in Attiya et al. [2006], where it was shown that it can be implemented wait-free using registers. This construction is a variant of the *deterministic* splitter introduced by Lamport [1987] for mutual exclusion, and was first used for deterministic renaming by Moir and Anderson [1995]. Figures 15 and 16 give intuitive descriptions for the structure of deterministic and randomized splitters, respectively.

A.2. The RatRace Adaptive Renaming Algorithm

Description. The algorithm is based on a binary tree structure, of unbounded height. Each node v in this tree contains a randomized splitter object RS_v. Each randomized splitter RS_v has two pointers, referring to randomized splitter objects corresponding to the left and right children of node v. Thus, if node v has children ℓ (left) and r (right), the left pointer of RS_v will refer to RS_ℓ, while the right pointer refers to RS_r. Any

process p_i returning left from the randomized splitter RS_v will call the split procedure of RS_ℓ, while processes returning right will call the split procedure of RS_r.

Processes start at the root node of the primary tree, and proceed left or right (with probability 1/2) through the tree until first returning stop at the randomized splitter RS_v associated to some node v. We say that a process *acquires* a randomized splitter s if it returns stop at the randomized splitter s. Once it acquires a randomized splitter, the process stops going down the tree. We will show that every process reaches depth at most $O(\log k)$ in the tree before acquiring a name, with high probability, and that every process returns, with probability 1.

Decision. Each process that acquires a randomized splitter in the primary tree returns the label of the corresponding node in a breadth-first search labelling of the primary tree.

Properties. The RatRace renaming algorithm ensures the following properties. The proof follows in a straightforward manner from the analysis for the test-and-set version of RatRace [Alistarh et al. 2010]. We provide a short proof here for completeness.

Name uniqueness follows since no two processes may stop at the same randomized splitter, as shown in Attiya et al. [2006]. We now provide a probabilistic upper bound on namespace size.

PROPOSITION 1 (RatRace RENAMING). *For $c \geq 3$ constant, the* RatRace *renaming algorithm described previously yields an adaptive renaming algorithm ensuring a namespace of size $O(k^c)$ in $O(\log k)$ steps, both with high probability in k. Every process eventually returns with probability 1.*

PROOF. Pick a process p, and assume that the process reaches depth d in the binary tree without acquiring a randomized splitter. By the properties of the randomized splitter, and by the structure of the algorithm, this implies that there exists (at least) one other process q which follows exactly the same path through the tree as process p. Necessarily, q must have made the same random choices as process p, at every randomized splitter on the path.

Let k be the number of participants in the execution, and pick $d = c \log k$, where $c \geq 3$ is a constant. The probability that an arbitrary process makes exactly the same $c \log k$ random choices as p is $(1/2)^{c \log k} = (1/k)^c$. By the union bound, the probability that there exists another process q which makes the same choices as p is at most $(k-1)(1/k)^c \leq (1/k)^{c-1}$. Applying the union bound again, we obtain that the probability that there exists a process p which takes more than $c \log k$ steps is at most $(1/k)^{c-2}$. This also implies that every process returns a name between 1 and k^c with probability $1 - (1/k)^c$. The termination bound follows by the same argument, by taking $d \to \infty$. □

REFERENCES

Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. 1993. Atomic snapshots of shared memory. *J. ACM* 40, 4, 873–890.

Yehuda Afek, Hagit Attiya, Arie Fouren, Gideon Stupp, and Dan Touitou. 1999. Long-lived renaming made adaptive. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 91–103. DOI:http://dx.doi.org/10.1145/301308.301335

Yehuda Afek, Eli Gafni, John Tromp, and Paul M. B. Vitányi. 1992. Wait-free test-and-set. In *Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG)* (Extended Abstract). Springer-Verlag, 85–94.

Yehuda Afek and Michael Merritt. 1999. Fast, wait-free (2k-1)-renaming. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 105–112. DOI:http://dx.doi.org/10.1145/301308.301338

Miklos Ajtai, Janos Komlós, and Endre Szemerédi. 1983. An $O(n \log n)$ sorting network. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing (STOC)*. ACM, 1–9. DOI: http://dx.doi.org/10.1145/800061.808726

Dan Alistarh and James Aspnes. 2011. Sub-logarithmic test-and-set against a weak adversary. In *Proceedings of the 25th International Conference on Distributed Computing (DISC)*. Springer-Verlag, 97–109.

Dan Alistarh, James Aspnes, Keren Censor-Hillel, Seth Gilbert, and Morteza Zadimoghaddam. 2011a. Optimal-time adaptive strong renaming, with applications to counting. In *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 239–248.

Dan Alistarh, James Aspnes, Seth Gilbert, and Rachid Guerraoui. 2011b. The complexity of renaming. In *Proceedings of the 52nd IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE, 718–727.

Dan Alistarh, Hagit Attiya, Seth Gilbert, Andrei Giurgiu, and Rachid Guerraoui. 2010. Fast randomized test-and-set and renaming. In *Proceedings of the 24th International Conference on Distributed Computing (DISC)*. Springer-Verlag, 94–108. http://portal.acm.org/citation.cfm?id=1888781.1888794

Dan Alistarh, Hagit Attiya, Rachid Guerraoui, and Corentin Travers. 2012. Early-deciding renaming in $O(\log f)$ rounds or less. In *Proceedings of the 19th International Colloquium on Structural Information and Communication Complexity (SIROCCO'12)*. Springer-Verlag.

James H. Anderson and Mark Moir. 1997. Using local-spin k-exclusion algorithms to improve wait-free object implementations. *Distrib. Comput.* 11, 1, 1–20. DOI: http://dx.doi.org/10.1007/s004460050039

James Aspnes, Hagit Attiya, and Keren Censor. 2012a. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM* 59, 1, 2:1–2:24.

James Aspnes, Hagit Attiya, Keren Censor-Hillel, and Faith Ellen. 2012b. Faster than optimal snapshots (for a while): Preliminary version. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC'12)*. ACM, New York, 375–384. DOI: http://dx.doi.org/10.1145/2332432.2332507

James Aspnes, Maurice Herlihy, and Nir Shavit. 1994. Counting networks. *J. ACM* 41, 5, 1020–1048.

Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Ruediger Reischuk. 1990. Renaming in an asynchronous environment. *J. ACM* 37, 3, 524–548.

Hagit Attiya and Vita Bortnikov. 2002. Adaptive and efficient mutual exclusion. *Distrib. Comput.* 15, 3, 177–189.

Hagit Attiya and Taly Djerassi-Shintel. 2001. Time bounds for decision problems in the presence of timing uncertainty and failures. *J. Parall. Distrib. Comput.* 61, 8, 1096–1109.

Hagit Attiya and Arie Fouren. 2001. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.* 31, 2, 642–664. DOI: http://dx.doi.org/10.1137/S0097539700366000

Hagit Attiya and Danny Hendler. 2010. Time and space lower bounds for implementations using k-CAS. *IEEE Trans. Parall. Distrib. Syst.* 21, 2, 162 –173. DOI: http://dx.doi.org/10.1109/TPDS.2009.60

Hagit Attiya, Danny Hendler, and Philipp Woelfel. 2008. Tight RMR lower bounds for mutual exclusion and other problems. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing (STOC)*. ACM, 217–226. DOI: http://dx.doi.org/10.1145/1374376.1374410

Hagit Attiya, Maurice Herlihy, and Ophir Rachman. 1995. Atomic snapshots using lattice agreement. *Distrib. Comput.* 8, 3, 121–132. DOI: http://dx.doi.org/10.1007/BF02242714

Hagit Attiya, Fabian Kuhn, C. Greg Plaxton, Mirjam Wattenhofer, and Roger Wattenhofer. 2006. Efficient adaptive collect using randomization. *Distributed Computing* 18, 3, 179–188. DOI: http://dx.doi.org/10.1007/s00446-005-0143-6

Hagit Attiya and Jennifer Welch. 1998. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics*. McGraw-Hill.

Amotz Bar-Noy and Danny Dolev. 1989. Shared-memory vs. message-passing in an asynchronous distributed environment. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 307–318. DOI: http://dx.doi.org/10.1145/72981.73003

Michael A. Bender and Seth Gilbert. 2011. Mutual Exclusion with $O(\log^2 \log n)$ Amortized work. In *Proceedings of the 52nd IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE, 728–737.

Elizabeth Borowsky and Eli Gafni. 1993. Immediate atomic snapshots and fast renaming. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 41–51. DOI: http://dx.doi.org/10.1145/164051.164056

Alex Brodsky, Faith Ellen, and Philipp Woelfel. 2006. Fully-adaptive algorithms for long-lived renaming. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*. 413–427.

James E. Burns and Gary L. Peterson. 1989. The ambiguity of choosing. In *PODC'89: Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*. ACM, New York, 145–157. DOI: http://dx.doi.org/10.1145/72981.72991

Armando Castañeda and Sergio Rajsbaum. 2010. New combinatorial topology bounds for renaming: the lower bound. *Distrib. Comput.* 22, 5–6, 287–301.

Armando Castañeda and Sergio Rajsbaum. 2012. New combinatorial topology bounds for renaming: The upper bound. *J. ACM* 59, 1, 3.

Soma Chaudhuri, Maurice Herlihy, and Mark R. Tuttle. 1999. Wait-free implementations in message-passing systems. *Theoret. Comput. Sci.* 220, 1, 211–245.

Bogdan S. Chlebus and Dariusz R. Kowalski. 2008. Asynchronous exclusive selection. In *PODC'08: Proceedings of the 27th ACM Symposium on Principles of Distributed Computing*. ACM, New York, 375–384. DOI: http://dx.doi.org/10.1145/1400751.1400801

Stephen A. Cook, Cynthia Dwork, and Rüdiger Reischuk. 1986. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM J. Comput.* 15, 1, 87–97.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* 3rd Ed. The MIT Press.

Edgser W. Dijkstra. 1965. Solution of a problem in concurrent programming control. *Commun. ACM* 8, 9, 569. DOI: http://dx.doi.org/10.1145/365559.365617

Wayne Eberly, Lisa Higham, and Jolanta Warpechowska-Gruca. 1998. Long-lived, fast, waitfree renaming with optimal name space and high throughput. In *Proceedings of DISC*. 149–160.

Alan David Fekete. 1990. Asymptotically optimal algorithms for approximate agreement. *Distrib. Comput.* 4, 9–29.

Faith Ellen Fich, Danny Hendler, and Nir Shavit. 2005. Linear lower bounds on real-world implementations of concurrent objects. In *Proceedings of the 46th IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE, 165–173.

Eli Gafni. 2009. The extended BG-simulation and the characterization of *t*-resiliency. In *Proceedings of the 41st ACM Symposium on Theory of Computing*. ACM, 85–92.

Wojciech Golab, Lisa Higham, and Philipp Woelfel. 2011. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the 43rd ACM Symposium on Theory of Computing (STOC)*. ACM, 373–382. DOI: http://dx.doi.org/10.1145/1993636.1993687

Wojciech M. Golab, Vassos Hadzilacos, Danny Hendler, and Philipp Woelfel. 2007. Constant-RMR implementations of CAS and other synchronization primitives using read and write operations. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. 3–12.

Jae-heon Yang and James H. Anderson. 1995. A fast, scalable mutual exclusion algorithm. *Distrib. Comput.* 9, 51–60.

Maurice Herlihy. 1991. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.* 13, 1, 123–149.

Maurice Herlihy and Nir Shavit. 1999. The topological structure of asynchronous computability. *J. ACM* 46, 6, 858–923.

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.* 12, 3, 463–492.

Prasad Jayanti. 1998. A time complexity lower bound for randomized implementations of some shared objects. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 201–210. DOI: http://dx.doi.org/10.1145/277697.277735

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

Yong-Jik Kim and James H. Anderson. 2012. A time complexity lower bound for adaptive mutual exclusion. *Distrib. Comput.* 24, 6, 271–297.

Donald E. Knuth. 1998. *The Art of Computer Programming*, Volume 3: Sorting and Searching 2nd Ed. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA.

Shay Kutten, Rafail Ostrovsky, and Boaz Patt-Shamir. 2000. The Las-Vegas processor identity problem (how and when to be unique). *J. Algor.* 37, 2, 468–494.

Leslie Lamport. 1987. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.* 5, 1, 1–11. DOI: http://dx.doi.org/10.1145/7351.7352

Richard J. Lipton and Arvin Park. 1990. The processor identity problem. *Inf. Process. Lett.* 36, 2, 91–94. DOI: http://dx.doi.org/10.1016/0020-0190(90)90103-5

Nancy A. Lynch. 1996. *Distributed Algorithms*. Morgan-Kaufmann.

Mark Moir and James H. Anderson. 1995. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Program.* 25, 1, 1–39. DOI: http://dx.doi.org/10.1016/0167-6423(95)00009-H

Mark Moir and Juan A. Garay. 1996. Fast, long-lived renaming improved and simplified. In *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG)*. Springer-Verlag, 287–303.

Michael Okun. 2010. Strong order-preserving renaming in the synchronous message passing model. *Theoret. Comput. Sci.* 411, 40–42, 3787–3794.

Alessandro Panconesi, Marina Papatriantafilou, Philippas Tsigas, and Paul M. B. Vitányi. 1998. Randomized naming using wait-free shared variables. *Distrib. Comput.* 11, 3, 113–124.

Marshall Pease, Robert Shostak, and Leslie Lamport. 1980. Reaching agreement in the presence of faults. *J. ACM* 27, 2, 228–234.

John Tromp and Paul Vitányi. 2002. Randomized two-process wait-free test-and-set. *Distrib. Comput.* 15, 3, 127–135. DOI: http://dx.doi.org/10.1007/s004460200071