Optimal-Time Adaptive Strong Renaming, with Applications to Counting

[Extended Abstract]

Dan Alistarh [*]
EPFL

James Aspnes [†]
Yale University

Keren Censor-Hillel [‡]
MIT

Seth Gilbert
NUS

Morteza Zadimoghaddam
MIT

ABSTRACT

We give two new randomized algorithms for strong renaming, both of which work against an adaptive adversary in asynchronous shared memory. The first uses repeated sampling over a sequence of arrays of decreasing size to assign unique names to each of n processes with step complexity $O(\log^3 n)$. The second transforms any sorting network into a strong adaptive renaming protocol, with an expected cost equal to the depth of the sorting network. Using an AKS sorting network, this gives a strong adaptive renaming algorithm with step complexity $O(\log k)$, where k is the contention in the current execution. We show this to be optimal based on a classic lower bound of Jayanti. We also show that any such strong renaming protocol can be used to build a monotone-consistent counter with logarithmic step complexity (at the cost of adding a max register) or a linearizable fetch-and-increment register (at the cost of increasing the step complexity by a logarithmic factor).

1. INTRODUCTION

The availability of unique names, or identifiers, is a fundamental prerequisite for efficiently solving a variety of problems in distributed systems. In some settings unique names are available, but come from a very large, practically unbounded namespace, which reduces their usefulness. Thus, the renaming problem, in which a set of processes are assigned distinct names from a small namespace, is one of the fundamental problems in distributed computing, and a significant amount of research, e.g. [1–7], studied its solvability and complexity in fault-prone distributed systems.

Renaming deterministically in the presence of crash faults can be expensive, and there are inherent limitations on the size of the achievable namespace. In particular, tight deterministic renaming, where the size of the namespace is exactly n, the size of the set of processes, is known to be impossible [1–3], and even the best known *loose* renaming algorithms, which relax the tight namespace requirement, have total step complexity at least $\Theta(n^2)$ [8, 9].

On the other hand, randomized solutions to renaming, e.g. [10–12], are known to be able to circumvent these limitations. The general strategy behind these algorithms is the following: we start from a list of randomized test-and-set objects, implemented using e.g. [10, 12, 13], and associate each test-and-set object with a unique name, which is usually its index in the list. By winning a certain test-and-set, a process acquires the associated name. In the simplest such algorithm [4, 11], a process starts at the head of the list and competes in test-and-set objects of increasing index, until it acquires a name. This simple strategy ensures a tight namespace, and is also *adaptive*, in that the size of the resulting namespace depends on the number of participants k, not on the maximum number of processes n. However, this algorithm has linear step complexity[1] in the number of participating processes.

Other, more complex strategies have been used, e.g. [10, 12], however the existence of a renaming algorithm that achieves a strong adaptive namespace using step complexity less than linear has remained an open problem. One of the challenges in building such algorithms is that each process has to acquire a unique name without probing linearly through the namespace, even though a large portion of the identifiers may be already taken. Moreover, the algorithm has to work in spite of a strong adversary, which may adjust concurrency and failures dynamically. Even worse, in the case of *adaptive* algorithms, the size of the namespace is not known initially, and has to be adjusted to match exactly the size of the set of participants.

Contribution. In this paper, we present two new randomized algorithms for strong renaming, both of which work against a strong adaptive adversary, and have poly-logarithmic step complexity, with high probability.

Our first algorithm, called BitBatching, is a strong renaming algorithm that allows process to find a single available test-and-set among n such objects using $O(\log^2 n)$ random probes, with high probability. To accomplish this, we start from a vector of n randomized test-and-set objects, which we partition into segments of decreasing size $n/2$, $n/4$, etc.,

[*] Supported by the SNF MICS Project.
[†] Supported in part by NSF grant CCF-0916389.
[‡] Supported by the Simons Postdoctoral Fellows Program.

PODC 2011, San Jose USA

[1] In the following, by step complexity we always mean *local*, per process, step complexity.

down to $\Theta(\log n)$. Each process attempts to grab $\Theta(\log n)$ randomly chosen test-and-set objects in each segment sequentially; if it fails to win one of these objects, it proceeds to the next segment. We prove that if a process makes it through all $O(\log n)$ segments without finding a free test-and-set, then, with high probability, all n objects have been acquired by the $n - 1$ other processes, which is impossible. The proof is based on a backward induction argument: we show that if a segment is full, then the previous segment must also be full with high probability. The algorithm is presented in Section 4.

Our second algorithm is the first to achieve a namespace that is both tight *and* adaptive in sub-linear time. The approach is different from those presented so far: we start from a sorting network [14] where the comparators are replaced with two-process test-and-set objects, which we call a *renaming network*, and prove that it solves strong adaptive renaming. The mechanism is that each process is assigned a distinct input port corresponding to its unique initial name, and follows a path through the network determined by leaving each comparator on its lower output wire if it wins the test-and-set, and on the upper output wire otherwise; the output name is the index of the output port it reaches. The expected step complexity of the algorithm is equal to the depth of the sorting network.

The procedure described above has the disadvantage that its complexity depends on the size of the initial namespace, since each process needs a distinct input port. We eliminate this limitation in Section 6, where we present a construction with unboundedly many ports, which maintains the properties of a sorting network when truncated to a finite number of input and output ports. In particular, when using an optimal AKS sorting network [15] as basis for our renaming network construction, we obtain an adaptive strong renaming algorithm whose step complexity is $O(\log k)$, in expectation, and $O(\log^2 k)$, with high probability.

We show that this algorithm is optimal in terms of time complexity in Section 7 by adapting a lower bound of Jayanti on the wakeup problem [16]. We prove that, for any c, any adaptive strong renaming algorithm that terminates with probability c has worst-case step complexity $\Omega(c \log k)$. The lower bound holds even if test-and-set objects with unit cost are available.

We find that being able to assign unique consecutive identifiers with logarithmic cost also has applications to counting. In Section 8.1, the strong adaptive algorithm is used to implement a monotone-consistent counter with $O(\log k)$ step complexity. To increment, a process requests a new name, and then writes it to a max-register, implemented in logarithmic time using the construction from [17]. To read the counter, a process simply returns the value of the max-register. Our counter implementation is more efficient by a logarithmic factor than the best previously known [17], but only guarantees monotone consistency, not linearizability.

We also show how to implement a linearizable fetch-and-increment object from any strong adaptive renaming protocol. In Section 8.2, we obtain a linearizable m-valued fetch-and-increment with $O(\log k \log m)$ cost, which can be shown to be optimal within a logarithmic factor by the same lower bound technique. The lower bound in Section 7 shows that this implementation is optimal within a logarithmic factor.

Discussion. Our results reveal a connection between sorting networks, adaptive strong renaming, and distributed counting, and provide tight bounds for adaptive tight randomized renaming.

The impossibility of wait-free strong renaming [1–3] is circumvented since we use randomization. There exist infinite length executions, in which the algorithms do not terminate, however these occur with probability 0.

The renaming network construction and the resulting counter and fetch-and-increment implementations can be made deterministic with no loss in terms of step complexity if two-process test-and-set or compare-and-swap objects with unit cost are available in hardware.

The efficient counting upper bounds require renaming implementations that are tight and locally efficient, so they cannot be obtained from previous renaming upper bounds. Also, note that our adaptive tight algorithm supersedes the BitBatching algorithm since it is adaptive and has better step complexity; however, the latter is superior in terms of space complexity.

We use AKS sorting networks [15] as the basis of our renaming networks in order to achieve optimal time complexity; however, these networks are known to be impractical [14]. Since our results hold for any sorting networks, an alternative would be to use constructible networks such as bitonic networks [14]; this trades constructibility for a logarithmic increase in running time.

Our renaming algorithms also show a separation in terms of step complexity between randomized renaming and randomized consensus. The results in [18] imply a lower bound of $\Omega(n)$ on the step complexity of randomized consensus, while we achieve randomized renaming in $O(\log n)$ steps per process, i.e. exponentially faster.

Due to space limitations, we only present proof sketches for some of our results, and some proofs are omitted. A full version of the paper with complete proofs can be found in [19].

2. MODEL AND PROBLEM STATEMENT

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

Problem Statement. The *renaming problem* [1] requires that each correct process should eventually return a name, and that the names returned should be unique. The size of the resulting namespace should only depend on n and on t. In the case of the *adaptive* renaming problem, the size of the namespace should depend on k, the contention in the current execution. The *tight* renaming problem requires that the size of the namespace be exactly n, while the *adaptive tight* renaming problem requires that the resulting namespace be of size exactly k. The complexity of our solutions is measured in terms of process steps (reads and writes, including random coin flips; we count all coin flips between two shared memory operations as one step). Therefore by step complexity we mean the number of steps a process takes during an execution. Total step complexity is the total number of process steps in an execution. Since atomic test-and-set operations are available on most modern machines, we state some of the complexity upper bounds also counting test-and-set operations as having unit cost.

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

In our algorithms, we use probabilistic implementations of test-and-set objects. It is known that such objects are implementable in asynchronous shared memory using randomization [13]. The implementation we use for n-process test-and-set is that of [12], which is adaptive to total contention; more precisely, the number of steps per process required by a test-and-set operation is $O(\log^2 k)$ w.h.p., where k is the number of participating processes. For 2-process test-and-set, we use the algorithm by Tromp and Vitányi [20], which has expected step complexity $O(1)$, and $O(\log n)$ step complexity with probability at least $1 - 1/n^c$ for $c > 1$ constant. We say that a process *wins* a test-and-set if it returns 1 from the object; otherwise, the process *loses* the test-and-set. A test-and-set is *acquired* if it has been won by a process.

3. RELATED WORK

The renaming problem was introduced in [1] by Attiya et al., where the authors also introduce a wait-free solution using $(2n - 1)$ names in an asynchronous message-passing system, and show that at least $(n+1)$ names are required in the wait-free case. The lower bound on the size of the namespace for deterministic solutions was improved to $(2n - 2)$ in a landmark paper by Herlihy and Shavit [2], later improved by Rajsbaum and Castañeda [3]. Adaptive renaming has been shown to be related to the set agreement task by Gafni et al. [21].

Considerable research has analyzed the complexity of deterministic implementations, e.g. [5–7, 9, 22–24]. For a detailed description of the deterministic results, we refer the reader to [12, 24]. Note that the deterministic lower bound on the namespace size does not apply in the case of our protocols since the "bad" executions in the lower bound occur with a probability that goes to 0 as the protocol takes more steps.

The first paper to analyze randomized renaming in an asynchronous system is by Panconesi et al. [10]. The authors present a non-adaptive solution ensuring a namespace of size $(1+\epsilon)n$, for $\epsilon > 0$ constant, with expected $O(M \log^2 n)$ total step complexity, where M is the size of the initial namespace. Their strategy is to introduce a new test-and-set implementation, and to assign names to processes based on the index of the test-and-set object they acquire. Along the same vein, Eberly et al. [11] obtained *tight* non-adaptive randomized renaming based on the test-and-set by Afek et al. [13]. Their implementation has $O(n \log n)$ amortized step complexity per process, under a given cost measure. The average-case total step complexity of their algorithm is $\Omega(n^3)$.

A recent paper by Alistarh et al. [12] introduced an *adaptive* test-and-set implementation with logarithmic step complexity called RatRace. They use this object to obtain a non-adaptive tight algorithm with $O(n \operatorname{polylog} n)$ total step complexity, and an adaptive loose algorithm with poly-logarithmic step complexity. Our strong adaptive algorithm uses the randomized splitter tree idea to reduce the size of the namespace to polynomial in k. This procedure has first been analyzed in [28] in the context of collect algorithms.

Compared to previous work, our algorithms achieve a tight namespace using only logarithmic step complexity.

We derive an $\Omega(\log k)$ lower bound on the time complexity of adaptive randomized renaming using the lower bound on the wakeup problem by Jayanti [16]. To the best of our knowledge, this is the only non-trivial lower bound on the time complexity of randomized renaming.

Monotone consistency has been introduced in reference [17], where the authors also present max-register and counter implementations with sub-linear time complexity. Their counter is deterministic and linearizable, and has complexity $O(\log^2 n)$ for polynomially many increments. Our counter implementation has complexity $O(\log n)$ in expectation, but is only monotone-consistent, and not linearizable.

Counting networks, introduced in [25], are other shared objects related to sorting networks. However, since their aim is to balance the number of processes exiting on the output ports, their applications and structure are in general different than those of renaming networks. 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. [26] to follow from earlier results of Aspnes et al. [25]; this is equivalent to our use of sorting networks for non-adaptive renaming in Section 5.

4. A NON-ADAPTIVE ALGORITHM

In this section, we present an algorithm that renames into n names, using polylogarithmic operations per process, with high probability. The algorithm assigns unique names to processes by repeatedly sampling over batches of test-and-set bits of decreasing size.

Description. The n processes share a vector of n test-and-set objects, each implemented using the RatRace algorithm [12]. For simplicity, we will assume that $n = 2^\kappa$, for an integer κ. We partition the vector of n test-and-set objects in *batches* as follows. Let $\ell = \lfloor \log(n/\log n) \rfloor$. For $1 \leq i < \ell$, batch B_i consists of vector positions from $n(2^{i-1} - 1)/2^{i-1} + 1$ to $n(2^i - 1)/2^i$. In particular, batch B_1 consists of the first half of the vector (from left to right), batch B_2 consists of the next quarter, and so on. Batch B_ℓ, which does not follow the above formula, consists of positions from $n(2^{\ell-1} - 1)/2^{\ell-1} + 1$ to position n. For $1 \leq i < \ell$, the length of batch B_i is $n/2^i$. Batch B_ℓ has length between $\log n$ and $2 \log n$ (see Figure 1 for an illustration).

Given this partitioning of the vector, processes (sequentially) compete in $O(\log n)$ test-and-set objects in each batch, starting from batch number 1 up to batch ℓ, and stopping when they first win a test-and-set object. More precisely, we define two *stages* in the algorithm. In the first stage, for every $1 \leq i < \ell$, each process p (sequentially) competes in $3 \log n$ *randomly chosen* test-and-set objects from every

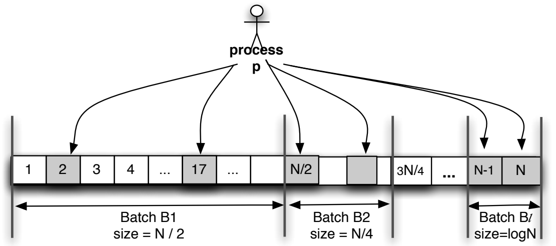

Figure 1: The BitBatching algorithm. The process makes $\Theta(\log n)$ random trials in each batch, until it first wins a test-and-set object.

batch B_i. If it did not stop before entering batch B_ℓ, the process competes in *every* test-and-set object in this batch. If the process finishes competing in batch B_ℓ and still did not win a test-and-set, then it enters the second stage, where it competes in all test-and-set objects from 1 to n, in sequence, from left to right. In the following, we will show that, with high probability, every process wins a test-and-set while in the first stage.

Analysis. The termination property holds with probability 1, by the properties of test-and-set, and by the structure of the algorithm. The name uniqueness property follows since no two processes may win the same test-and-set. In the following, we prove upper bounds on the step complexity of the algorithm.

The first lemma shows that, with high probability, every process gets a name while doing the first pass through the test-and-set vector.

LEMMA 1 (LOCAL TRIALS). *With high probability, every process terminates while executing the first stage, i.e. it returns a name after competing in $O(\log^2 n)$ test-and-set objects.*

PROOF. Assume that there exists a process p that competes in test-and-set objects in all batches (B_i) for $i = 1, \ldots, \ell$ without winning any test-and-set objects. In particular, this implies that p has competed in all the test-and-set objects in batch B_ℓ. Since p did not win any test-and-set in this batch, it follows that this batch is already "full," i.e. there are at least $\log n$ distinct processes that won each of the test-and-set objects in this batch (we consider the linearization order at each of these objects). Let S_ℓ be this set of processes.

It follows that each of the processes in S_ℓ has performed $2 \log n$ random trials in the batch $B_{\ell-1}$, and did not succeed in acquiring a name in this batch. We prove the following claim.

CLAIM 1. *Fix $i \geq 1$, and let B_i be a batch. Let S be a set of at least $|B_i|/2$ processes that perform $3 \log n$ random trials each in batch B_i, and* none *succeeds in winning a test-and-set in batch B_i. Then the batch B_i is* full *with high probability, i.e. there exists a set of processes F_i with $|F_i| = |B_i|$ such that each process in F_i won a distinct test-and-set object in batch B_i. Moreover, $S \cap F_i = \emptyset$.*

PROOF. Given that $|B_i|$ processes performed $3 \log n$ random trials each in the batch B_i, it follows that the batch B_i has been tried by a total of at least $3/2 \cdot |B_i| \log n \geq 3/2 \cdot |B_i| \cdot \log |B_i|$ random probes. Since all these probes have been already scheduled by the adversary, we can apply a standard coupon collector analysis [27] to conclude that, w.h.p., *each* test-and-set has been tried by one of the $3/2 \cdot |B_i| \cdot \log |B_i|$ random probes. Since none of these probes succeeds, by the linearizability of test-and-set, it follows that each of the test-and-set objects in batch B_i has been won by a distinct process, w.h.p., which concludes the proof of the claim. □

Returning to the proof, by Claim 1, we obtain that batch $B_{\ell-1}$ is full with high probability. The processes occupying batches B_ℓ and $B_{\ell-1}$ have tried for $3 \log n$ times each in batch $B_{\ell-2}$, without success. In this case, Claim 1 implies that batch $B_{\ell-2}$ is full w.h.p. By backward induction over the batch number, we obtain that all batches $(B_i)_{i \in \{\ell, \ldots, 1\}}$ are full w.h.p. By the union bound, it follows that the vector is full with high probability. Since the algorithm guarantees that a process may win a single test-and-set object, it follows that, if process p moves to stage two of the algorithm, then w.h.p. there are at least $n + 1$ participating processes in this execution, which contradicts the definition of n.

Using this bound on the number of trials, we obtain bounds on the local and total step complexity of our algorithm, by carefully bounding the maximum number of processes that access a test-and-set object.

COROLLARY 1. *With high probability, every process returns after $O(\log^3 n \log \log n)$ local steps. The expected step complexity is $O(\log^2 n \log \log n)$.*

COROLLARY 2. *With high probability, the total step complexity of the algorithm is $O(n \log^2 n \log \log n)$. The expected total step complexity is $O(n \log n \log \log n)$. The total number of test-and-set operations performed in an execution is $O(n \log n)$ with high probability.*

5. RENAMING USING A SORTING NETWORK

In this section, we present a solution for adaptive strong renaming using a sorting network. For simplicity, we first describe the solution when the bound on the size of the initial namespace, M, is finite and known. We circumvent this limitation in Section 6.

Description. We start from an arbitrary sorting network of size M, in which we replace the comparator modules with two-process test-and-set objects to obtain a *renaming network.* We use the renaming network to solve adaptive strong renaming as follows. Each participating process enters the execution on the input wire in the sorting network corresponding to its unique initial value. The process competes in two-process test-and-set instances as follows: if the process *wins* a two-process test-and-set, then it moves "up" in the network; otherwise it moves "down." The process continues until it reaches an output port. The process returns the index of the output port as its output value.

Analysis. In the following, we show that the renaming network construction solves adaptive strong renaming, i.e. 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. In particular, if we use the optimal AKS sorting network [15] as the basis for the renaming network, then we solve strong renaming using expected $O(\log M)$ step complexity. In Section 6.1, we show how to use this idea to obtain an adaptive strong renaming algorithm with $O(\log k)$ step complexity, which is optimal, as per Section 7.

Theorem 1. *The renaming network construction solves strong adaptive renaming with probability* 1.

Proof (Sketch). Termination with probability 1 follows from the structure of the sorting network and from the termination property of the two-process test-and-set implementation [20].

We prove name uniqueness and namespace tightness in the following. 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 of these processes has assigned a unique input port i of 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 not in I.

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 (e.g., crashed processes and ghost processes). The next step is to simulate the output of these processes by extending the current renaming network execution. The rules for deciding the outputs of test-and-set objects for these processes are the following: if the current test-and-set already has a winner, i.e. a (distinct) process that goes "up", then the current process automatically goes "down" at this test-and-set. Otherwise, if the winner has not yet been decided, then we decide the test-and-set according to the input value corresponding to the two processes: the process with the smaller input value goes "up," while the other process goes "down." Test-and-set objects where both processes have the same corresponding value are decided arbitrarily.

In this way, we obtain an execution of a renaming network in which M processes participate, and each test-and-set object has a winner and a loser. Notice that we can re-order these test-and-set operations into *stages*, such that all test-and-set operations that may be performed in parallel are assigned to the same stage. 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, we obtain a sequence of comparator operations ordered in stages, in which each stage contains only comparison operations that may be performed in parallel. The key observation now is that all these test-and-set operations obey the comparison property of the comparators in a sorting network, 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."

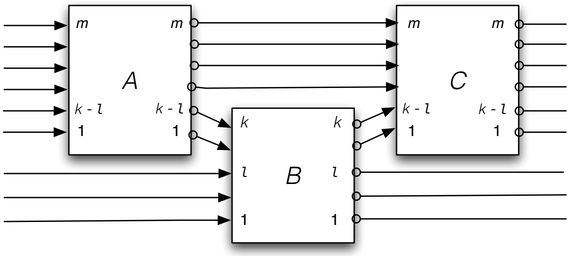

Figure 2: One stage in the construction of the adaptive sorting network. The original network B is "sandwiched" between the larger networks A and C.

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 no two input values may be sorted to the same output port, we first obtain that the output port values 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 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 $M - k$ processes that have not taken a step in $\mathcal{E}$ must be associated with the lower $M - k$ output ports of the sorting network. Conversely, processes that have returned a value in the execution $\mathcal{E}$ must have returned a value between 1 and k. □

We can then obtain an upper bound on the step complexity of the protocol when starting from an AKS sorting network [15]. The key observation is that that the number of test-and-set objects a process enters is bounded by the depth of the original sorting network.

Corollary 3. *The renaming network obtained from an AKS sorting network [15] with M inputs solves the strong adaptive renaming problem with M initial names. It guarantees termination with probability* 1 *and name uniqueness in all executions, using expected $O(\log M)$ local steps. The step complexity is $O(\log M \log n)$ w.h.p., and the total step complexity is $O(k \log M)$ w.h.p. for $M = \Theta(n)$, where k is the contention in the execution.*

6. STRONG ADAPTIVE RENAMING

In this section, we present an algorithm for adaptive strong renaming based on an adaptive sorting network construction. In particular, the algorithm works irrespective of the size of the initial namespace.

6.1 An Adaptive Sorting Network

We present a recursive construction of a sorting network of arbitrary size. We will guarantee that the resulting construction ensures 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.

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 non-uniform—different paths through the network have different lengths, with the lowest part of the sorting network having the same width as B.

Formally, let us suppose we have sorting networks A, B, and C, where A and C have width m and B has width k. 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 $\ell \le k/2$ and construct a new sorting network ABC with inputs $B_1, B_2, \ldots B_\ell, A_1, \ldots A_m$ and outputs $B'_1, B'_2, \ldots B'_m, A'_1, A'_2, \ldots A'_m$. 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 2). We now show that the resulting construction is a sorting network.

Lemma 2. *The network ABC constructed as described above is a sorting network.*

Proof. The proof uses the well-known Zero-One Principle [14]: 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 $B_1 \ldots B_\ell$ and $A_1 \ldots A_m$. 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 $B'_1 \ldots B'_{z_B+\max(k-\ell,z_A)}$. 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: $z_A \le k-\ell$. 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. The sorting network works 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 $\ell \le k/2$, $z_B + k - \ell \ge k/2 \ge \ell$, and B sends ℓ zeros out its direct outputs $B'_1 \ldots B_\ell$. All remaining zeros are fed into C, which sorts them to the next $z_A + z_B - \ell$ positions. Again the sorting network works. □

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, whose proof is immediate from the construction and Lemma 2.

Lemma 3. *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.*

Now let us show how to recursively construct a large sorting network with $\operatorname{polylog} N$ depth when truncated to the first N 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 [15], we have $c = 1$; for constructible networks (e.g., the bitonic sorting network [14]), we have $c = 2$.

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

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

Observe that $w_{k+1} = w_k^2$ and $d_{k+1} = 2a\log^c(w_k^2 - w_k/2) + d_k \le 4a\log^c w_k + d_k$. Solving these recurrences gives $w_k = 2^{2^k}$ and $d_k = \sum_{i=0}^{k} 2^{c(i+2)}a = O(2^{ck})$.

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

But we can in fact state something stronger:

Theorem 2. *Each of the networks S_k constructed above is a sorting network, with the property that any value that enters on the n-th input and leaves on the m-th output traverses $O(\log^c \max(n,m))$ comparators.*

Proof. That S_k is a sorting network follows from induction on k using Lemma 2. For the second property, let $S_{k'}$ be the smallest stage in the construction of S_k to which input n and output m are directly connected. Then $w_{k'-1}/2 < \max(n,m) \le w_{k'}/2$, which we can rewrite as $2^{2^{k'-1}} < 2\max(n,m) \le 2^{2^{k'}}$ or $k'-1 < \lg\lg\max(n,m) \le k'$, implying $k' = \lceil\lg\lg\max(n,m)\rceil$. By Lemma 3, the given value stays in $S_{k'}$, meaning it traverses at most $d_{k'} = O\left(2^{ck'}\right) = O\left(2^{c\lceil\lg\lg\max(n,m)\rceil}\right) = O\left(\lg^c\max(n,m)\right)$ comparators. □

6.2 Strong Adaptive Renaming Algorithm

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 arbitrarily large.

Description. Our algorithm is composed of two stages. In the first stage, each process obtains a unique temporary name in a namespace of size polynomial in k, with high probability. The algorithm, which we call TempName, is as follows. We allocate a binary tree of randomized splitters (as previously defined in [28]), of unbounded height. Each process starts the protocol at the root splitter in the tree; if it does not stop at the current splitter, it goes either left or right, each with probability $1/2$, until it manages to acquire a splitter. Notice that, by the properties of the splitter, the process will stop at height at most k in the tree. Once it stops at a splitter, the process adopts a temporary name corresponding to the index of the splitter in a breadth-first search labeling of the tree nodes. Variants of this algorithm have been previously analyzed in [12, 28].

In the second stage, we consider a renaming network as defined in Section 5, instantiated using the adaptive sorting network of Section 6.1. Let R be the resulting renaming network. Each process uses the temporary name it has acquired in the first stage as the index of its 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, our algorithm may not be wait-free if k is unbounded. In particular, 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.

Analysis. Before we proceed with the proof of the main theorem, please recall that the TempName algorithm has the following properties: (1) given k participating processes, it assigns names from 1 to k^c with probability $1 - 1/k^{c-1}$, where $c > 1$ is a constant; (2) its step complexity is $O(\log k)$ with high probability in k. The proof can be found in [12,28]. We now prove the following.

THEOREM 3. *For any finite $k > 0$, the adaptive renaming network construction based on the AKS sorting network solves adaptive strong renaming for k processes. The local time complexity of the protocol is $O(\log^2 k)$ with high probability, and $O(\log k)$ in expectation.*

PROOF. We first prove that the resulting construction solves adaptive strong renaming for any $k > 0$. First, we know that the temporary names obtained in stage one are between 1 and k^c, with high probability, for a constant $c \geq 1$. Therefore we will assume that, during the current execution, each process enters an input port of the renaming network between 1 and k^c. We will truncate the renaming network after the first k^c input ports. By Theorem 2, we obtain that the original comparison network truncated after the first k^c input ports is in fact a sorting network. From Theorem 1, we obtain that the second stage of the construction implements adaptive strong renaming for at most k^c processes. The first claim follows.

For the complexity bound, first recall that any process takes $O(\log k)$ steps during the first stage, with high probability. Second, from Theorem 2 we obtain that the number of test-and-sets a process competes in during an execution of the renaming network is $O(\log \max(\ell, m))$, where ℓ is the number of the input port for the process, and m is the number of the output port for the process. Notice that $\ell \leq k^c$, with high probability, and $m \leq k$, by the adaptive tight property of the renaming network.

Therefore a process competes in $O(\log k)$ test-and-set instances in the second stage. By the properties of the two-process test-and-set, we obtain that a process takes expected $O(\log k)$ steps in an execution, and at most $O(\log^2 k)$ steps, with high probability in k. □

7. LOWER BOUND

We prove that our adaptive renaming algorithm is optimal in terms of local time complexity starting from a lower bound for the wakeup problem. Recall that the *wakeup problem* for n processes [16] is specified as follows: (1) Every process terminates in a finite number of its steps, returning either 0 or 1, (2) In every run in which all processes terminate, at least one process returns 1, and (3) In every run in which one or more processes return 1, every process takes at least one step before any process returns 1. We start by re-stating the lower bound result by Jayanti.

THEOREM 4 (JAYANTI, [16]). *Consider any algorithm for the n-process wakeup problem in shared memory where only LL, SC, validate, move, and swap operations may be used. If the algorithm terminates with probability c, then its worst-case expected shared-access time complexity is at least $c \log n$.*

Based on this, we prove the following lower bound on adaptive strong renaming. Note that the lower bound holds even when test-and-set operations are available, and are assumed to have unit cost. Also, since shared-access time complexity as used in [16] is a lower bound on the (local) step complexity of the algorithm, we claim our lower bound for expected step complexity.

THEOREM 5 (LOWER BOUND). *Consider a randomized algorithm for adaptive strong renaming in asynchronous shared memory augmented with test-and-set operations, which terminates with probability c. Then the algorithm has worst-case expected step complexity $\Omega(c \log k)$, where k is the number of participating processes.*

PROOF (SKETCH). We assume for contradiction that there exists an algorithm A that solves adaptive strong renaming and terminates with probability c, which has worst-case expected time complexity $o(c \log k)$, for any k.

We first transform algorithm A from an algorithm using read, write, and test-and-set operations, to an algorithm that uses only LL, SC and move operations. (Recall that the move operation takes as arguments a register R and a value v, and changes the value of R to v atomically. It is essentially the same as a write operation in read-write shared memory. For a precise definition of the LL/SC and move operations, please see [16]). We first replace all registers and test-and-set bits with registers supporting LL/SC and move, initialized to $\perp$. Any read operation on a register is replaced with a LL operation on the corresponding register. Any write(v) operation on a register R is replaced with a move(R, v) operation on that register. Any test-and-set operation is replaced with a LL operation followed by a SC operation with value 1 on the same register. Clearly, this transforms algorithm A into an algorithm A' that uses only LL/SC and move operations, with a constant increase in time complexity.

We now consider the algorithm A' in a system where k, the number of participating processes, is fixed and known. We can use the algorithm A' to solve the wakeup problem as follows: if a process receives name k from A', then it returns 1. Otherwise, it returns 0. We now check that this solves the wakeup problem.

First, if every process terminates, then, by the strong adaptivity of the namespace, there has to exist a process that obtains name k and returns 1 in the wakeup problem. On the other hand, if a process p returns 1, then it has obtained name k, therefore, by the strong adaptivity of the namespace, there have to exist $k - 1$ other processes that took at least one step in this execution. (Otherwise, by indistinguishability, process p would have to return name $k-1$ or smaller.) Hence, this algorithm solves the wakeup problem in a system with k processes.

Termination is ensured with the same probability c, and the time complexity of the protocol is $o(c \log k)$. Therefore, we obtain that the wakeup problem can be solved in a system with k processes using $o(c \log k)$ local steps by an algorithm which terminates with probability c, contradicting Theorem 4. □

Based on the same rationale, we can obtain a lower bound for linearizable fetch-and-increment objects.

Corollary 4 (Fetch-and-increment). *Any fetch-and-increment object in asynchronous shared memory augmented with test-and-set operations which terminates with probability c has worst-case expected local time complexity $\Omega(c \log k)$, where k is the number of participating processes.*

8. APPLICATIONS TO COUNTING

8.1 A Monotone-Consistent Counter

We now build a monotone-consistent counter algorithm with logarithmic step complexity, based on the strong adaptive renaming algorithm.

Description. The processes share an adaptive renaming object implemented using the construction from Section 6.2, and a linearizable max register, implemented using the logarithmic construction from [17]. 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. For the read operation, the process simply reads the value of the max register and returns it.

Analysis. We now prove the properties of the counter.

Lemma 4 (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 $O(\min(\log v, O(n)))$.*

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 registers (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 the complexity of the first stage of the adaptive renaming protocol is $O(\log v)$, and the number of temporary names is $O($ poly $v)$ with high probability. It then follows that the complexity of the adaptive renaming object is $O(\log v)$ in expectation, and $O(\log^2 v)$ with high probability in v. 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. □

```
Shared: boolean doorway, initially open;

procedure $\ell$-test-and-set();
if $O.doorway = closed$ then
    return false
else
    $name \leftarrow$ tight-renaming();
    if $name \leq \ell$ then  return true
    else
        $O.doorway \leftarrow closed$
        return false
```

Algorithm 1: The ℓ-test-and-set implementation.

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

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

Algorithm 2: The ℓ-fetch-and-increment object.

Linearizability. We show a non-linearizable execution of our counter implementation. Consider three processes p_1, p_2, and p_3. Process p_2 obtains name 2 and writes it to the max register. After p_2's operation terminates, p_1 starts its increment operation and obtains name 1 from the renaming network and writes it to the max register (this 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. 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. Both read operations have to 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.

8.2 Linearizable Bounded-Value Fetch-and-Increment

In this section, we show how to use an adaptive strong renaming protocol to construct a linearizable m-valued fetch-and-increment object, i.e. 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. We first use the strong 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 strong 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 already started the operation earlier from winning. Algorithm 1 presents the pseudocode.

The second part of the m-valued fetch-and-increment construction is based on a recursive tree construction, whose pseudocode is presented in Algorithm 2. For simplicity, we present the construction when m is a power of two. (The construction for general m can be easily obtained from the construction for the smallest power of two larger than m, by returning $m-1$ instead of any value larger than $m-1$.) For $\ell \geq 1$, we build an ℓ-fetch-and-increment object out of (a) one $\ell/2$-test-and-set object, and (b) 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. The two children of a 1-fetch-and-increment are two 0-fetch-and-increment objects. We implement such an object with an empty data structure on which the fetch-and-increment operation always returns 0.

The construction starts at level m and unfolds to a tree, whose leaves are 0-valued fetch-and-increment objects. For each level ℓ at which it accesses the right fetch-and-increment child, the process adds the value $\ell/2$ in a local variable, and returns the final value of this variable.

Analysis. We start by precisely defining the ℓ-test-and-set object.

DEFINITION 1. *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.

We now show the correctness of our ℓ-test-and-set implementation. Intuitively, we show that exactly ℓ processes may get *true*, by the adaptivity and tightness of the namespace; any operation that starts later sees the doorway closed, and must therefore must return *false*.

LEMMA 5 (ℓ-TEST-AND-SET). *Procedure* ℓ*-test-and-set in Algorithm 1 implements a linearizable* ℓ*-test-and-set object with expected step complexity* $O(\log k)$.

PROOF. By the correctness of the adaptive strong renaming algorithm, ℓ processes obtain a name whose value is at most m, and therefore exactly ℓ processes return *true*. For 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 non-overlapping 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 non-overlapping operations, we only need to argue about non-overlapping operations in different categories, since within each category this order is preserved by construction. Let op_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 the above implements a linearizable ℓ-test-and-set object. □

We conclude with a proof of correctness of the fetch-and-increment implementation. The basic idea is that the linearizability of the $\ell/2$-test-and-set object allows us to linearize any operation incrementing to value v before any operation incrementing to value $v' > v$. The complexity bound follows from the construction.

THEOREM 6 (m-FETCH-AND-INCREMENT). *The* m*-fetch-and-increment implementation in Algorithm 2 is linearizable, and has step complexity* $O(\log k \log m)$ *in expectation, and* $O(\log^2 k \log m)$ *with high probability.*

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 non-overlapping operations, we need to argue only about non-overlapping 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. □

9. CONCLUSIONS AND FUTURE WORK

In this paper, we introduce new randomized algorithms for adaptive strong renaming which work against a strong adaptive adversary. Our upper bound in the strong adaptive case is time-optimal, and shows a connection between sorting networks, renaming, and distributed counting. In particular, it can be used to obtain a monotone-consistent counter implementation with logarithmic complexity, and a linearizable fetch-and-increment implementation with almost-optimal polylogarithmic complexity.

The renaming network technique and the resulting counter implementations can be made deterministic with no loss in terms of step complexity if two-process test-and-set or compare-and-swap objects are available in hardware, which is common on modern machines.

One immediate direction of future work is to see if our techniques can be used to obtain a *linearizable* counter implementation with optimal logarithmic cost. A more general direction would be to see whether we can use the connection between counting, renaming and sorting to obtain lower bounds for counting or renaming from sorting lower bounds. A third direction would be to try to apply our techniques to other problems, such as *long-lived* renaming [24], resource allocation, or mutual exclusion.

10. ACKNOWLEDGEMENTS

We would like to thank Hagit Attiya, Rachid Guerraoui and Prasad Jayanti for useful discussions and support. We would also like to thank the anonymous reviewers for many useful comments.

11. REFERENCES

[1] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk, "Renaming in an asynchronous environment," *Journal of the ACM*, vol. 37, no. 3, pp. 524–548, 1990.

[2] M. Herlihy and N. Shavit, "The topological structure of asynchronous computability," *J. ACM*, vol. 46, no. 2, pp. 858–923, 1999.

[3] A. Castañeda and S. Rajsbaum, "New combinatorial topology upper and lower bounds for renaming," in *PODC '08: Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, (New York, NY, USA), pp. 295–304, ACM, 2008.

[4] J. H. Anderson and M. Moir, "Using local-spin k-exclusion algorithms to improve wait-free object implementations," *Distrib. Comput.*, vol. 11, no. 1, pp. 1–20, 1997.

[5] M. Moir and J. H. Anderson, "Fast, long-lived renaming (extended abstract)," in *WDAG '94: Proceedings of the 8th International Workshop on Distributed Algorithms*, (London, UK), pp. 141–155, Springer-Verlag, 1994.

[6] M. Moir and J. A. Garay, "Fast, long-lived renaming improved and simplified," in *WDAG '96: Proceedings of the 10th International Workshop on Distributed Algorithms*, (London, UK), pp. 287–303, Springer-Verlag, 1996.

[7] M. Moir and J. A. Garay, "Fast, long-lived renaming improved and simplified," in *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, (New York, NY, USA), p. 152, ACM, 1996.

[8] B. S. Chlebus and D. R. Kowalski, "Asynchronous exclusive selection," in *PODC '08: Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, (New York, NY, USA), pp. 375–384, ACM, 2008.

[9] H. Attiya and A. Fouren, "Adaptive and efficient algorithms for lattice agreement and renaming," *SIAM J. Comput.*, vol. 31, no. 2, pp. 642–664, 2001.

[10] A. Panconesi, M. Papatriantafilou, P. Tsigas, and P. M. B. Vitányi, "Randomized naming using wait-free shared variables," *Distributed Computing*, vol. 11, no. 3, pp. 113–124, 1998.

[11] W. Eberly, L. Higham, and J. Warpechowska-Gruca, "Long-lived, fast, waitfree renaming with optimal name space and high throughput," in *DISC*, pp. 149–160, 1998.

[12] D. Alistarh, H. Attiya, S. Gilbert, A. Giurgiu, and R. Guerraoui, "Fast randomized test-and-set and renaming," in *DISC*, pp. 94–108, 2010.

[13] Y. Afek, E. Gafni, J. Tromp, and P. M. B. Vitányi, "Wait-free test-and-set (extended abstract)," in *WDAG '92: Proceedings of the 6th International Workshop on Distributed Algorithms*, (London, UK), pp. 85–94, Springer-Verlag, 1992.

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

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

[16] P. Jayanti, "A time complexity lower bound for randomized implementations of some shared objects," in *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '98, (New York, NY, USA), pp. 201–210, ACM, 1998.

[17] J. Aspnes, H. Attiya, and K. Censor, "Max registers, counters, and monotone circuits," in *PODC*, pp. 36–45, 2009.

[18] H. Attiya and K. Censor, "Tight bounds for asynchronous randomized consensus," *J. ACM*, vol. 55, no. 5, pp. 1–26, 2008.

[19] D. Alistarh, J. Aspnes, K. Censor-Hillel, S. Gilbert, and M. Zadimoghaddam, "Optimal-time adaptive tight renaming, with applications to counting." EPFL Technical Report, Jan. 2011.

[20] J. Tromp and P. Vitányi, "Randomized two-process wait-free test-and-set," *Distrib. Comput.*, vol. 15, no. 3, pp. 127–135, 2002.

[21] E. Gafni, A. Mostéfaoui, M. Raynal, and C. Travers, "From adaptive renaming to set agreement," *Theor. Comput. Sci.*, vol. 410, pp. 1328–1335, March 2009.

[22] J. E. Burns and G. L. Peterson, "The ambiguity of choosing," in *PODC '89: Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, (New York, NY, USA), pp. 145–157, ACM, 1989.

[23] E. Borowsky and E. Gafni, "Immediate atomic snapshots and fast renaming," in *PODC '93: Proceedings of the twelfth annual ACM symposium on Principles of distributed computing*, (New York, NY, USA), pp. 41–51, ACM, 1993.

[24] A. Brodsky, F. Ellen, and P. Woelfel, "Fully-adaptive algorithms for long-lived renaming," in *DISC*, pp. 413–427, 2006.

[25] J. Aspnes, M. Herlihy, and N. Shavit, "Counting networks," *J. ACM*, vol. 41, pp. 1020–1048, Sept. 1994.

[26] H. Attiya, M. Herlihy, and O. Rachman, "Efficient atomic snapshots using lattice agreement (extended abstract)," in *Proceedings of the 6th International Workshop on Distributed Algorithms*, WDAG '92, (London, UK), pp. 35–53, Springer-Verlag, 1992.

[27] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* New York, NY, USA: Cambridge University Press, 2005.

[28] H. Attiya, F. Kuhn, C. G. Plaxton, M. Wattenhofer, and R. Wattenhofer, "Efficient adaptive collect using randomization," *Distrib. Comput.*, vol. 18, no. 3, pp. 179–188, 2006.