The Distributed Computing Column

by

Stefan Schmid

TU Berlin
T-Labs, Germany
stefan.schmid@tu-berlin.de

The Renaming Problem: Recent Developments and Open Questions

Dan Alistarh
Microsoft Research

1 Introduction

1.1 The Renaming Problem

The theory of distributed computing centers around a set of fundamental problems, also known as *tasks*, usually considered in variants of the two classic models of distributed computation: *asynchronous shared-memory* and *asynchronous message-passing* [50]. These fundamental tasks are important because, in general, their computability and complexity in a given system model gives a good measure of the model's power.

In this article, we survey recent results and open questions regarding one of the canonical distributed tasks, called *renaming*. Simply put, in the renaming problem, a set of processes need to pick *unique names* from a *small namespace*. Intuitively, renaming can be seen as the *dual* of the classic *distributed consensus* problem [48]: if solving consensus means that processes need to *agree* on a single value, renaming asks participants to *disagree* in a constructive way, by each returning a *distinct* value from a small space of options.

More formally, the renaming problem assumes that processes start with unique initial names from a large, virtually unbounded namespace,[1] 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 $M > 0$, which is a parameter given in advance. The namespace size M should only depend on n, the maximum number of participating processes.

The *adaptive* version of renaming requires the size of the namespace M 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 range of the namespace matches exactly the number of participating processes, renaming is

[1] In the absence of unique initial identifiers, it is known that renaming is impossible [49].

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; later, we will see that *strong* renaming algorithms can in fact be used to implement other distributed objects, such as counters and mutual exclusion.

The reader may now want to pause and briefly consider how to solve this problem. One natural idea is for each participant to pick names *at random* between 1 and M. Assuming we have a way of handling name collisions (usually done through auxiliary *test-and-set* or *splitter* objects, which we describe later), processes may simply re-try new random names until successful. Notice however that the relationship between n, the number of participants, and M, the range of available names, critically influences the complexity of this procedure. If M is much larger than n, for instance $M \geq n^2$, then, by standard analysis, choices will almost never collide, and therefore each completes within a constant number of trials. If $M = Cn$, for $C > 1$ constant, then the probability of a collision is *constant* < 1 in each trial, and therefore each participant will complete within $O(\log n)$ trials, with high probability. A particularly interesting case is when $M = n$, i.e. we want a *tight* namespace. In this case, it appears likely that at least one process will have to try a large fraction of the names before succeeding, i.e. run for *linear* time. For this unlucky participant, this strategy is no better than trying out all names sequentially, in some order.

The basic intuition above can be turned, with some care, into working renaming algorithms [10]. It also sugests that there is a trade-off between the size of the namespace we wish to achieve, and the complexity of our algorithm. In the following, we will see that this trade-off is somewhat slanted in favor of randomization: we are able to attain a *tight* namespace in logarithmic worst-case *expected time*, but the (deterministic) worst-case running time for renaming is linear, even for large namespace size.

Before we delve into the details of these results, let us first cover some historical background. The renaming problem was formally introduced more than 25 years ago [17]. A significant amount of research, e.g. [25, 29, 51, 42, 3, 20, 37, 54], 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 using only read-write registers [42, 30]. In fact, $(n + t - 1)$ is the best achievable namespace size when t processes may crash [30, 31]. The proof of this result is very interesting, and quite complex, as it requires the use of complex topological techniques [42]. As for consensus, 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, e.g. [37].

The *complexity* of renaming has also been the focus of significant research effort, e.g. [54, 27, 51, 52, 3, 20, 37, 33, 1]. In particular, much of this work considered the shared-memory model, perhaps due to the simpler way to express the time complexity of an algorithm. However, in spite of this effort, until recently, no time optimality results were known for shared-memory renaming, either for randomized or deterministic algorithms.

1.2 Recent Developments

In the following, we will survey a recent series of papers [10, 6], giving tight bounds for the *time complexity* of renaming in asynchronous shared-memory.[2] Our survey covers some of the results from these papers, and adopts their notation and presentation for the technical content.

Specifically, for deterministic algorithms, reference [6] gave a *linear* lower bound on the time complexity of renaming into any namespace of sub-exponential size. This bound can be matched by previously known algorithms. e.g. [51, 52]. (See Section 3 for a detailed discussion.) For randomized algorithms, [6] gave tight logarithmic upper and lower bounds on the time complexity of adaptive renaming into a namespace of *linear* size. Together, these results give an exponential time complexity separation between deterministic and randomized implementations of renaming.

It is also interesting to study connections between renaming and implementations of other shared objects. Since renaming can be solved trivially using objects with stronger semantics, such as stacks, queues, or counters supporting fetch-and-increment, lower bounds for renaming also apply to these widely-used, practical objects. Thus, the above results can be used to match or improve the previously known lower bounds for these problems (see Table 1 for an overview), but also to obtain efficient implementations of more complex shared objects. Due to space constraints, we refer the reader to [6] for the latter constructions.

Conceptually, the improved upper and lower bounds are based on new connections between renaming and other fundamental objects: sorting networks [45] and mutual exclusion [36]. Specifically, the first step is a construction showing that sorting networks can be used to obtain optimal-time solutions for *strong adaptive* randomized renaming. Further, it can be shown that the resulting algorithm can be extended to an efficient solution to mutual exclusion.

To obtain the linear lower bound on deterministic renaming, we can re-trace the previous argument: we start from a known linear lower bound on the time complexity of mutual exclusion, and derive by reduction a lower bound on renaming.

[2] In this model, time is measured in terms of number of *steps*, that is, shared-memory operations performed by a processor until completion.

Shared Object	Lower Bound	Type	Matching Algorithms	New Result
Deterministic ck-renaming	$\Omega(k)$ $\Omega(k \log(k/c))$	Local Global	[52] -	Yes Yes
Randomized ck-renaming	$\Omega(k \log(k/c))$	Global	Section 5	Yes
c-Approximate Counter	$\Omega(k \log(k/c))$	Global	[15]	Yes
Fetch-and-Increment	$\Omega(k)$ $\Omega(k \log k)$	Local Global	[51] Section 5	Improves on [39] Improves on [21]
Queues and Stacks	$\Omega(k)$ $\Omega(k \log k)$	Local Global	[41] -	Improves on [39] Improves on [21]

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

The lower bound on the time complexity of randomized renaming follows from a separate information-based argument.

2 Model and Problem Statements

2.1 The Asynchronous Shared Memory Model

In this section, we introduce the asynchronous shared memory model [24], [50] , and the cost measures we will use for the analysis of algorithms.

Asynchronous Shared Memory. We consider the standard asynchronous shared-memory model, in which a set of n processes $\Pi = \{p_1, \ldots, p_n\}$ can communicate through operations on shared multi-writer multi-reader atomic registers. We will denote by k the *contention* in an execution, i.e. the actual number of processes that take steps in the execution. Obviously, $k \leq n$ throughout.

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 paper, we assume this upper bound is $t = n - 1$.) A *failed* process does not execute any further instructions. A process that does not crash during an execution is *correct*.

Identifiers. 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.

Wait-Freedom. An algorithm is *wait-free* if it ensures that every method call by a correct process returns within a finite number of steps [43]. Throughout this paper, we will consider wait-free algorithms.

Schedules and Adversaries. 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 (generally considered to be the number of steps). 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. 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, different assumptions on the relation between the scheduler and the random coin flips that processes perform during an execution may lead to different results. We will assume that the adversary controlling the schedule is the standard *strong* adversary, which 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).

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.

2.2 Problem Statements

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

Renaming. The *renaming problem*, first introduced in [17], 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 [17], 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 paper, 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, i.e. $T = n$. 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 *non-adaptive* renaming problem.

```
Variable:;
Value, a binary atomic register,
initially 0;
procedure test-and-set();
    if Value = 0 then
        Value ← 1;
        return 0;
    else
        return 1;
```

Figure 2: 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;
    else
        return s;
```

Figure 3: Sequential specification of the compare-and-swap object.

Test-and-Set. The *test-and-set* object, whose sequential specification is given in Figure 2, 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.

Compare-and-swap. The *compare-and-swap* object can be seen a generalization of test-and-set, whose underlying register supports multiple values (as opposed to only 0 and 1). Its sequential specification is presented in Figure 3. 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 $s \neq oldV$.

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.

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 [24].

- *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 paper, we focus on shared-memory mutual exclusion algorithms. As for renaming, there exists a distinction between adaptive and non-adaptive solutions. A classical, non-adaptive, 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 A Brief History of Renaming

Message-passing Models. The renaming problem, defined in Section 2.2, was introduced by Attiya et al. [17], in the asynchronous message-passing model. The paper presented a non-adaptive 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. [32] 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*, i.e. two processes may distinguish their states only through comparison operations. Attiya and Djerassi-Shintel [19] 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 [53] 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 [38]. Alistarh et al. [11] analyzed Okun's algorithm and showed that it is also *early-deciding*, i.e. 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. Recent work in the same model [12] has shown that a expected $O(\log \log n)$ time can be obtained using randomization.

Returning to the asynchronous message-passing model, Alistarh, Gelashvili, and Vladu [13] recently gave a randomized solution which solves tight renaming in $O(\log^2 n)$ rounds and $O(n^2)$ messages. They also show that this message complexity is optimal.

Shared-Memory Models. The first shared-memory renaming algorithm was given by Bar-Noy and Dolev [25], who ported the synchronous message-passing algorithm of Attiya et al. [17] 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 [29] 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 [17].) In a landmark paper, Herlihy and Shavit [42] used algebraic topology to show that there exist values of n for which wait-free $(2n - 2)$-renaming is impossible. Recently, Castañeda and Rajsbaum [30], [31] gave a full characterization, proving 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 [49], [46] 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 non-zero probability.

Later work focused on the time-namespace size trade-off. Moir and Anderson appear to be the first to use deterministic splitters to solve renaming [51]. Afek and Merritt [3] presented an adaptive read-write renaming algorithm with optimal namespace of size $(2k-1)$, and $O(k^2)$ step complexity. Attiya and Fouren [20] gave an adaptive $(6k - 1)$-renaming algorithm with $O(k \log k)$ step complexity. Chlebus and Kowalski [33] gave an adaptive $(8k - \log k - 1)$-renaming algorithm with $O(k)$ step complexity. For *long-lived* adaptive renaming, there exist implementations

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

with $O(k^2)$ time complexity for renaming into a namespace of size $O(k^2)$, e.g. [1]. The fastest such algorithm with optimal $(2k - 1)$ namespace size has $O(k^4)$ step complexity [20].

The time lower bound in Section 6 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 [51]. 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 [51] also present implementations of renaming from registers supporting set-first-zero and bitwise-and operations. In this paper, the authors also notice the fact that adaptive tight renaming can solve mutual exclusion. (This connection is also mentioned in [18].) Using load-linked and store-conditional primitives, Brodsky et al. [28] 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 [14], 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. [54]. They presented a non-adaptive 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.

A second paper to analyze randomized renaming was by Eberly et al. [37]. The authors obtain a *strong* non-adaptive renaming algorithm based on the randomized wait-free test-and-set implementation by Afek et al. [2]. 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. [10] generalized the approach by Panconesi et al. [54] 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 non-adaptive tight algo-

rithm with $O(nn)$ 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. Recent work by Alistarh, Aspnes, Giakkoupis and Woelfel [8] showed that, if we allow the algorithm to break the namespace requirement with some probability, then we can solve renaming in expected $O(\log \log n)$ time, by using a multi-level random choice strategy. This strategy can also be extended to adaptive algorithms, with a similar running time.

The renaming network algorithm presented in this paper first appeared in [7]. It is the first algorithm to achieve strong adaptive renaming in sub-linear time, improving exponentially on the time complexity of previous solutions. The same paper 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. [22] to follow from earlier results of Aspnes et al. [16]; this is equivalent to our use of sorting networks for non-adaptive renaming in Section 5.1.1. The lower bounds in this paper first appeared in [9].

Recent work also looked into the space complexity of this problem. Reference [40] gives linear lower bounds using a novel version of covering arguments, while reference [35] gave the first space-optimal renaming algorithm, which uses only $O(n)$ registers.

4 Renaming Building Blocks

In this section, we illustrate some of the main building blocks developed for renaming algorithms by way of example. We present a randomized algorithm which renames into a namespace of size polynomial in k, with logarithmic step complexity in expectation. This algorithm can also be extended to solve adaptive test-and-set [10], and will be a useful sub-routine for achieving a tight namespace in logarithmic time. We focus on the structure of the algorithm; its proof of correctness follows from the original analysis [10].

4.1 Deterministic and Randomized Splitters

The *deterministic splitter* object, was introduced by Lamport to solve mutual exclusion efficiently in the absence of contention [47]. This object, whose structure is given in Figure 5, provides the following semantics.

- Every correct process returns either stop, left, or right.

[4] In the following, by n we denote $\log^c n$, for some integer $c \geq 1$.

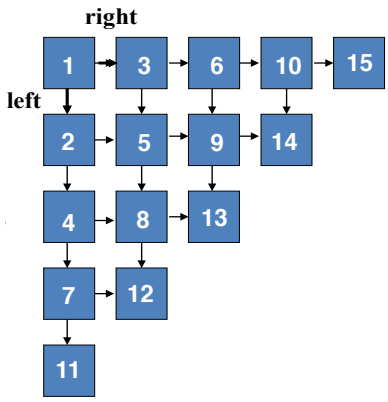

Figure 4: Structure and labeling of a deterministic splitter network.

- At most one process returns stop.
- If a single correct process calls split, then it returns stop.
- In an execution in which $k \geq 1$ processes access the object, at most $k-1$ processes return left, and at most $k-1$ processes return right.

A very interesting use of the deterministic splitter is in the context of the renaming problem: Moir and Anderson [51] noticed that splitters connected in a rectangular grid, as depicted in Figure 4, solve renaming.

More precisely, the key property of the splitter is that it changes direction for at least one of the calling processes, they show that a single process may access at most $k-1$ distinct splitter objects in the grid before returning stop at one of these objects. Given a labelling of the splitters as in Figure 4, each process may return the label of the splitter it returned stop from as its new name. A simple analysis yields that the names returned are from 1 to k^2.

The *randomized splitter* object is a weak synchronization primitive which allows a process to *acquire* it if is 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 [23], where it was shown that it can be implemented wait-free using registers. Next, we will see how splitters can be used to solve renaming in expected logarithmic time.

4.2 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 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. The key property of this process is that, in an execution with k participants, each 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 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 [10]. We provide a short proof here for completeness.

Name uniqueness follows since no two processes may stop at the same randomized splitter, which is one of the basic properties of this object [23]. We now provide a probabilistic upper bound on namespace size.

[RatRace Renaming] For $c \geq 3$ constant, the RatRace renaming algorithm described above yields an adaptive renaming algorithm ensuring a namespace of size

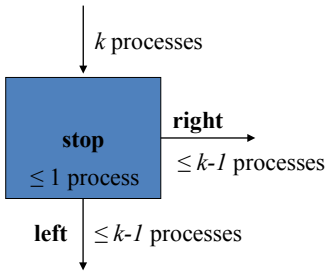

Figure 5: Deterministic splitter.

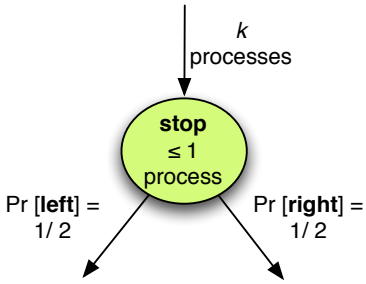

Figure 6: Randomized splitter.

$O(k^c)$ 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 $d \to \infty$. □

5 Adaptive Strong Renaming in Logarithmic Expected Time

In the previous section, we have seen a way of obtaining a namespace that is polynomial in the number of participants k, in logarithmic time. We now give a way of tightening the namespace to an optimal one, of size k, while preserving logarithmic running time. Logarithmic time is in fact optimal [6].

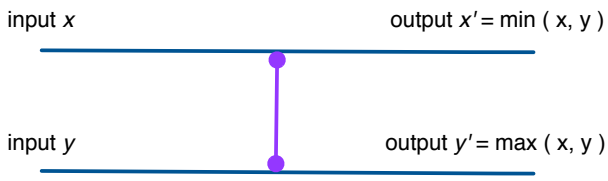

Figure 7: Structure of a comparator.

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 (running a loose renaming algorithm such as the one from the previous section), 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.[5]

[5]Notice that, if the contention k is small, the failure probability $O(1/k^c)$ with $c \geq 2$ constant may be non-negligible. 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 ← input wire corresponding to v_i in R;
    while w is not an output wire do
        T ← next test-and-set on wire w of R;
        res ← T.test − and − set( );
        if res = 0 then
            w ← output wire x′ of T;
        else
            w ← output wire y′ of T;
    return w.index;
```

Figure 8: Pseudocode for executing a renaming network.

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 7 (please see standard texts, e.g. [34], 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 8 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, i.e. follows output port x' of the test-and-set; otherwise it

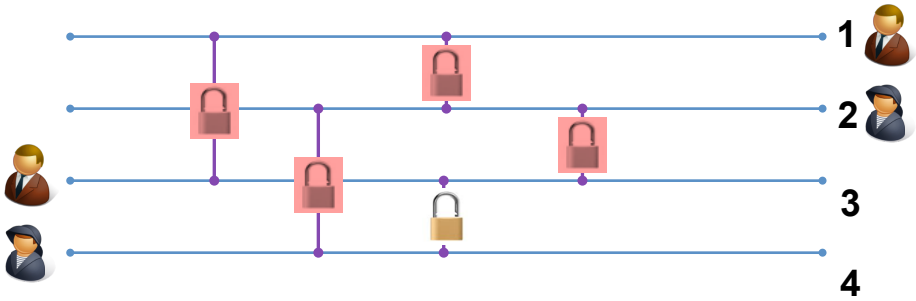

Figure 9: Execution of a renaming network. The two processes start at arbitrary distinct input ports, and proceed through the network until reaching an output port. The two-process test-and-set objects are depicted as locks. A two process test-and-set object is highlighted if it has been won during the 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.

moves "down," i.e. 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 9 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 [55]; in Section 6, 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, 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.

Theorem 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 below), 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*, i.e. that no two processes may access the same port of a two-process test-and-set object. 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 [34]. 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, i.e. 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 [55], the network guarantees termination with probability 1. We prove name uniqueness and namespace tightness by ensuring the following claim.

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, i.e. 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 $\mathcal{E}$, i.e. 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," i.e. 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 5.1.2, the execution is well-formed, i.e. 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. The above 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 5.1.2, 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, i.e. 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 $P \setminus C$, i.e. 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 [4]. (Recall that the AKS construction [4] 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.)

[AKS] The renaming network obtained from an AKS sorting network [4] 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 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 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 non-uniform—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 corre-

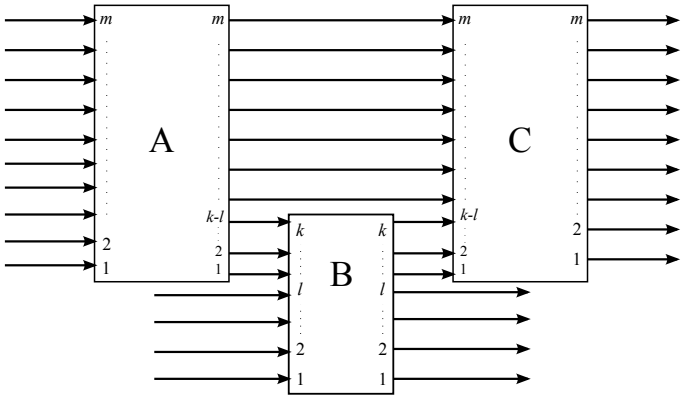

Figure 10: One stage in the construction of the adaptive sorting network. The small labels indicate port number: upper is higher.

sponding inputs of C: outputs $A'_{k-\ell+1}, \ldots, A'_m$ are wired to inputs $C_{k-\ell+1}, \ldots, C_m$. (See Figure 10.)

Lemma 1. *The network ABC constructed as described above is a sorting network.*

Proof. The proof uses the well-known Zero-One Principle [34]: 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: $z_A \leq k - \ell$. 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 $B'_1 \ldots B_\ell$. All remaining zeros are fed into C, which sorts them to the next $z_A + z_B - \ell$ positions. Again, the sorting network 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 2. *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 1; v does not enter A initially, and is sorted to one of the output $B'_1 \dots B'_\ell$, meaning that it also avoids C. □

Now let us show how to recursively construct a large sorting network with 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 [4], we have $c = 1$ and very large a; for constructible networks (e.g., the bitonic sorting network [45]), 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 1, 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 2. *For any $j \geq 0$, the network S_j 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_j is a sorting network follows from induction on j using Lemma 1.

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 $j' = \lg \lg \max(n, m)$. By Lemma 2, the given value stays in $S_{j'}$, meaning it traverses at most $d_{j'} = O\left(2^{cj'}\right) = O\left(2^{c \lg \lg \max(n,m)}\right) = O\left(\lg^c \max(n, m)\right)$ comparators. □

5.2.2 Transformation to a Renaming Nework

We now apply the previous results to renaming networks.

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 S_j has at least M inputs and M outputs. By Theorem 2, 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 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 [4] 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 [45]. On the other hand, since the construction accepts any sorting network as basis, we can use Batcher's bitonic sorting network [45], 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 3 (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 [55], 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 5.2.2, 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, i.e. 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 5.2.2, we obtain that a process performs $O(\log M)$ test-and-set accesses in any execution. Since the randomized test-and-set construction of [55], 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 [10], 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.

[RatRace Renaming] For $c \geq 3$ constant, the RatRace renaming algorithm described above 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.

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 3 with RatRace renaming. We obtain a correct adaptive strong renaming algorithm.

For the upper bounds on complexity, by Proposition 5.2.4, 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 3, 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.

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.

6 From an Optimal Randomized Algorithm to a Tight Deterministic Lower Bound

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 3. We first prove the lower bound for *adaptive* deterministic renaming, and then extend it to *non-adaptive* renaming by reduction. The lower bound will hold for algorithms that either rename into a sub-exponential 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 sub-exponential 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 $O(\log M(k))$.[6]

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 Anderson and Kim [44]. When plugging in any sub-exponential function for $M(k)$ in the expression bounding the RMR complexity of the adaptive mutual exclusion algorithm $ME(T(R))$, we obtain that the algorithm R must have step complexity at least linear in k.

[6]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.

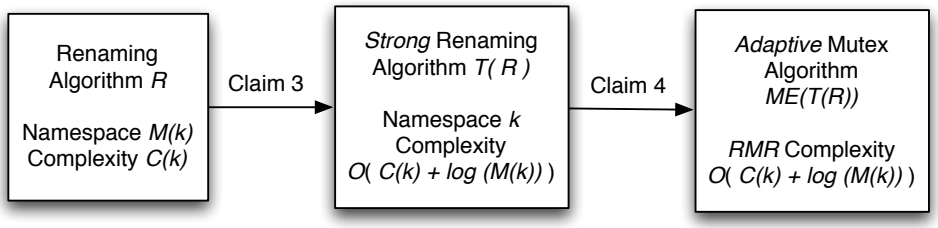

Figure 11: The structure of the reduction in Theorem 4.

Applications. This result also implies a linear lower bound on the time complexity of *non-adaptive* renaming algorithms, which guarantee names from 1 to some polynomial function in n, with n known. This generalization holds by reduction, and is proven in full in [6].

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.

6.1 Adaptive Lower Bound

In this section, we prove the following result.

Theorem 4 (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 5.2.4.

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).

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 3, 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*, i.e. 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 4, 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 Anderson and Kim [44, Theorem 2], stated below.

Theorem 5 (Mutex Time Lower Bound [44])**.** *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 [44].*

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 above mutual exclusion lower bound contradicts the existence of algorithm $ME(T(R))$. The contradiction arises from our initial assumption on the existence of algorithm R. The claim about step complexity follows since, for wait-free algorithms, the RMR complexity is always a lower bound on step complexity. The claim about the number of rename operations follows from the structure of the transformation and from that of the mutual exclusion lower bound of [44], in which each process performs the entry section once. □

6.1.1 Technical Notes

Relation between k and n. The lower bound of Anderson and Kim [44] 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 above, 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, 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, e.g. [52], [7], 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 Anderson and Kim [44] applies to all mutex algorithms ensuring livelock-freedom, and our transformation does not require a strengthening of this progress condition.

6.2 Applications

6.2.1 Non-Adaptive Renaming

The above argument can be extended to apply to *non-adaptive* renaming algorithms as well, as long as they start with names from a namespace of *unbounded* size, which matches the problem definition we considered. The argument is technical, and requires the definition of an auxiliary task called renaming *with fails*, which allows for the possibility of failure when acquiring a name. We refer the reader to [6] for the complete argument, and simply state the claim here.

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

6.2.2 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 3. *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.

[Queues, Stacks, Fetch-and-Increment] Consider a wait-free linearizable 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. 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 Discussion and Open Questions

We have surveyed tight bounds on the complexity of the renaming problem in asynchronous shared-memory, both for deterministic and randomized algorithms. In particular, we have seen that, using randomization, we can achieve a tight namespace in logarithmic expected time, and that deterministic implementations of renaming have linear time complexity as long as they ensure a polynomial-size namespace.

Several open questions remain. In shared-memory, the 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 [33] matches the linear time lower bound, giving a namespace of size $(8k - \log k - 1)$; an elegant algorithm by Attiya and Fouren [20] 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 et al. [3], with time complexity $O(k^2)$. We have thus reached our first open question.

Q1. What are the trade-offs between time complexity and namespace size for deterministic asynchronous renaming?

One disadvantage of the renaming network algorithm is that it is based on an AKS sorting network [4], which has prohibitively high constants hidden inside the asymptotic notation [45]. 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 non-adaptive algorithms exist, which could in theory go below the logarithmic threshold. We conjecture that $\Omega(\log n)$ steps is a lower bound for non-adaptive randomized algorithms as well.

Q2. What are the *tight* bounds on the time complexity of randomized non-adaptive renaming?

One aspect of these concurrent data structures, which has been somewhat neglected by research is *space* complexity, i.e. the number of registers necessary for correct shared-memory implementations. Recent work [35, 40] has begun looking into this area as well. The question of tight bounds for renaming parametrized by namespace size is still open however, and should yield interesting new insights on this problem.

Q3. What are the space-time-namespace trade-offs for renaming?

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 objects there do not exist algorithms that match this logarithmic lower bound. In terms of circumventing this bound, recent results [5], [26] suggest that weaker adversarial models and relaxing object semantics, e.g. allowing approximate implementations, could be used to go below this logarithmic threshold.

Q4. Give tight bounds for asynchronous queues, stacks, and counters.

An area where open questions are still abundant is that of *message-passing* implementations. In particular, the round complexity of renaming is open in both synchronous and asynchronous models, and known techniques do not appear to apply in this setting. Recent work [13] has given tight quadratic bounds for the message complexity of renaming in the classic asynchronous model, but the question of time (round) complexity of renaming in this model is still open.

Q5. What is the time complexity of renaming in asynchronous message-passing?

References

[1] Yehuda Afek, Hagit Attiya, Arie Fouren, Gideon Stupp, and Dan Touitou. Long-lived renaming made adaptive. In *Proc. 18th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 91–103. ACM, 1999.

[2] Yehuda Afek, Eli Gafni, John Tromp, and Paul M. B. Vitányi. Wait-free test-and-set (extended abstract). In *Proc. 6th International Workshop on Distributed Algorithms (WDAG)*, pages 85–94. Springer-Verlag, 1992.

[3] Yehuda Afek and Michael Merritt. Fast, wait-free (2k-1)-renaming. In *Proc. 18th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 105–112. ACM, 1999.

[4] Miklos Ajtai, Janos Komlós, and Endre Szemerédi. An $O(n \log n)$ sorting network. In *Proc. 15th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1–9. ACM, 1983.

[5] Dan Alistarh and James Aspnes. Sub-logarithmic test-and-set against a weak adversary. In *Proc. 25th International Conference on Distributed Computing (DISC)*, pages 97–109, 2011.

[6] Dan Alistarh, James Aspnes, Keren Censor-Hillel, Seth Gilbert, and Rachid Guerraoui. Tight bounds for asynchronous renaming. *J. ACM*, 61(3):18:1–18:51, June 2014.

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

[8] Dan Alistarh, James Aspnes, George Giakkoupis, and Philipp Woelfel. Randomized loose renaming in o(log log n) time. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 200–209, New York, NY, USA, 2013. ACM.

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

[10] Dan Alistarh, Hagit Attiya, Seth Gilbert, Andrei Giurgiu, and Rachid Guerraoui. Fast randomized test-and-set and renaming. In *Proc. 24th International Conference on Distributed Computing (DISC)*, pages 94–108. Springer-Verlag, 2010.

[11] Dan Alistarh, Hagit Attiya, Rachid Guerraoui, and Corentin Travers. Early-Deciding Renaming in O(log f) Rounds or Less. In *Proc. 19th International Colloquium on Structural Information and Communication Complexity (SIROCCO '12)*. Springer-Verlag, 2012.

[12] Dan Alistarh, Oksana Denysyuk, Luís Rodrigues, and Nir Shavit. Balls-into-leaves: Sub-logarithmic renaming in synchronous message-passing systems. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 232–241, New York, NY, USA, 2014. ACM.

[13] Dan Alistarh, Rati Gelashvili, and Adrian Vladu. How to elect a leader faster than a tournament. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 365–374, New York, NY, USA, 2015. ACM.

[14] James H. Anderson and Mark Moir. Using local-spin k-exclusion algorithms to improve wait-free object implementations. *Distributed Computing*, 11(1):1–20, 1997.

[15] James Aspnes, Hagit Attiya, and Keren Censor. Polylogarithmic concurrent data structures from monotone circuits. *Journal of the ACM*, 59(1):2:1–2:24, February 2012.

[16] James Aspnes, Maurice Herlihy, and Nir Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, September 1994.

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

[18] Hagit Attiya and Vita Bortnikov. Adaptive and efficient mutual exclusion. *Distributed Computing*, 15(3):177–189, 2002.

[19] Hagit Attiya and Taly Djerassi-Shintel. Time bounds for decision problems in the presence of timing uncertainty and failures. *J. Parallel Distrib. Comput.*, 61(8):1096–1109, 2001.

[20] Hagit Attiya and Arie Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31(2):642–664, 2001.

[21] Hagit Attiya and Danny Hendler. Time and space lower bounds for implementations using k-cas. *IEEE Trans. Parallel and Distrib. Syst.*, 21(2):162 –173, 2010.

[22] Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distributed Computing*, 8(3):121–132, March 1995.

[23] Hagit Attiya, Fabian Kuhn, C. Greg Plaxton, Mirjam Wattenhofer, and Roger Wattenhofer. Efficient adaptive collect using randomization. *Distributed Computing*, 18(3):179–188, 2006.

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

[25] Amotz Bar-Noy and Danny Dolev. Shared-memory vs. message-passing in an asynchronous distributed environment. In *Proc. 8th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 307–318. ACM, 1989.

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

[27] Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *Proc. 12th Annual ACM symposium on Principles of Distributed Computing (PODC)*, pages 41–51. ACM, 1993.

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

[29] James E. Burns and Gary L. Peterson. The ambiguity of choosing. In *PODC '89: Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 145–157, New York, NY, USA, 1989. ACM.

[30] Armando Castañeda and Sergio Rajsbaum. New combinatorial topology bounds for renaming: the lower bound. *Distributed Computing*, 22(5-6):287–301, 2010.

[31] Armando Castañeda and Sergio Rajsbaum. New combinatorial topology bounds for renaming: The upper bound. *Journal of the ACM*, 59(1):3, 2012.

[32] Soma Chaudhuri, Maurice Herlihy, and Mark R. Tuttle. Wait-free implementations in message-passing systems. *Theor. Comput. Sci.*, 220(1):211–245, 1999.

[33] Bogdan S. Chlebus and Dariusz R. Kowalski. Asynchronous exclusive selection. In *PODC '08: Proceedings of the twenty-seventh ACM symposium on*

Principles of distributed computing, pages 375–384, New York, NY, USA, 2008. ACM.

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

[35] Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Leslie Lamport. Adaptive register allocation with a linear number of registers. In Yehuda Afek, editor, *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, volume 8205 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2013.

[36] Edgser W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569–, September 1965.

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

[38] Alan David Fekete. Asymptotically optimal algorithms for approximate agreement. *Distributed Computing*, 4:9–29, 1990.

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

[40] Maryam Helmi, Lisa Higham, and Philipp Woelfel. Space bounds for adaptive renaming. In *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, pages 303–317, 2014.

[41] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, January 1991.

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

[43] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[44] Yong-Jik Kim and James H. Anderson. A time complexity lower bound for adaptive mutual exclusion. *Distributed Computing*, 24(6):271–297, 2012.

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

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

[47] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, January 1987.

[48] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[49] Richard J. Lipton and Arvin Park. The processor identity problem. *Inf. Process. Lett.*, 36(2):91–94, October 1990.

[50] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[51] Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Program.*, 25(1):1–39, October 1995.

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

[53] Michael Okun. Strong order-preserving renaming in the synchronous message passing model. *Theor. Comput. Sci.*, 411(40-42):3787–3794, 2010.

[54] Alessandro Panconesi, Marina Papatriantafilou, Philippas Tsigas, and Paul M. B. Vitányi. Randomized naming using wait-free shared variables. *Distributed Computing*, 11(3):113–124, 1998.

[55] John Tromp and Paul Vitányi. Randomized two-process wait-free test-and-set. *Distributed Computing*, 15(3):127–135, 2002.