# Randomized versus Deterministic Implementations of Concurrent Data Structures

THÈSE N$^{O}$ 5447 (2012)

PRÉSENTÉE LE 6 AOÛT 2012

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DE PROGRAMMATION DISTRIBUÉE

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Dan ALISTARH

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2012

*To Ludmila*

# Abstract

One of the key trends in computing over the past two decades has been increased distribution, both at the processor level, where multi-core architectures are now the norm, and at the system level, where many key services are currently distributed over multiple machines. Thus, understanding the power and limitations of computing in a concurrent, distributed setting is one of the major challenges in Computer Science.

In this thesis, we analyze the complexity of implementing concurrent data structures in asynchronous shared memory systems. We focus on the complexity of a classic distributed coordination task called *renaming*, in which a set of processes need to pick distinct names from a small set of identifiers. We present the first tight bounds for the time complexity of this problem, both for deterministic and randomized implementations, solving a long-standing open problem in the field. For deterministic algorithms, we prove a tight linear lower bound; for randomized solutions, we provide logarithmic upper and lower bounds on time complexity. Together, these results show an exponential separation between deterministic and randomized renaming solutions. Importantly, the lower bounds extend to implementations of practical shared-memory data structures, such as queues, stacks, and counters.

From a technical perspective, this thesis highlights new connections between the distributed renaming problem and other fundamental objects, such as sorting networks, mutual exclusion, and counters. In particular, we show that sorting networks can be used to obtain optimal randomized solutions to renaming, and that, in turn, the existence of these solutions implies a linear lower bound on the complexity of the problem.

In sum, the results in this thesis suggest that deterministic implementations of shared-memory data structures do not scale well in terms of worst-case time complexity. On the positive side, we emphasize randomization as a natural alternative, which can circumvent the deterministic lower bounds with high probability. Thus, a promising direction for future work is to extend our randomized renaming techniques to obtain efficient implementations of concurrent data structures.

Keywords: concurrent algorithms, shared memory, data structures, renaming, randomization, lower bounds

# Résumé

Les deux dernières décennies ont vu l'augmentation de la distribution des processus s'affirmer comme une tendance majeure de l'évolution des systèmes informatiques. Ceci aussi bien au niveau des microprocesseurs, qui sont maintenant pratiquement tous multi-coeur, que au niveau système, où les services principaux sont maintenant répartis sur plusieurs machines. La compréhension des possibilités et des limitations du calcul réparti et concurrent est donc d'un intérêt majeur en informatique.

La présente thèse analyse la complexité algorithmique de l'implémentation de structures de données concurrentes dans les systèmes à mémoire partagée. Nous concentrons nos efforts sur la complexité d'une tâche de coordination classique dite du renommage. La tâche consiste à attribuer un nom distinct appartenant à un ensemble restreint de noms à chaque processus du système. Nous présentons les premières bornes optimales de la complexité temporelle de cette tâche pour les implémentations déterministes et probabilistes. Ces résultats résolvent un problème ouvert de longue date dans le domaine. Pour les algorithmes déterministes, nous établissons une borne inférieure linéaire. Quant aux algorithmes probabilistes, nous établissons des bornes inférieure et supérieures logarithmiques. Ces résultats montrent qu'il existe une différence exponentielle de complexité entre les solutions probabilistes et déterministes pour la tâche du renommage. La borne inférieure obtenue dans le cas déterministe s'applique aussi aux implémentations déterministes de structures de données telles que les piles, les files, et les compteurs.

D'un point de vue technique, cette thèse met en lumière des connections nouvelles entre la tâche du renommage et d'autres tâches fondamentales telles que les réseaux de tri, l'exclusion mutuelle, et les compteurs. En particulier, nous montrons qu'il est possible d'obtenir une solution probabiliste optimale à la tâche du renommage en utilisant les réseaux de tri. L'existence de cette solution implique une borne inférieure linéaire à la complexité du problème dans le cas déterministe.

En somme, les résultats obtenus lors de cette thèse suggèrent que les implémentations déterministes des structures de données en mémoire partagée ne s'adaptent pas bien à la montée en charge d'un système. De manière plus positive, nos résultats mettent en lumière l'utilisation des méthodes probabilistes, qui permettent de contourner ce problème avec grande probabilité. L'extension des techniques utilisées pour résoudre de manière probabiliste le problème du renommage aux structures de données concurrentes est donc une direction de recherche prometteuse.

Mots-clés: algorithmes concurrents, mémoire partagée, structures de données, renommage, méthodes probabilistes, bornes inférieures

# Preface

This thesis presents part of the Ph.D. work performed under the supervision of Prof. Rachid Guerraoui at the EPFL School of Computer and Communication Sciences. It focuses on the complexity of concurrent shared-memory data structures, in particular the complexity of the renaming problem. The main results of this thesis appeared originally in the following papers.

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

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

3. Dan Alistarh, Hagit Attiya, Seth Gilbert, Andrei Giurgiu, and Rachid Guerraoui. Fast randomized test-and-set and renaming. In *Proceedings of the 24th International Conference on Distributed Computing (DISC 2010)*, pages 94-108, 2010.

During my Ph.D., I was also involved in other aspects of distributed computing, such as shared-memory data structures, the complexity of speculative distributed algorithms, and efficient wireless communication in a hostile environment. This resulted in the following publications.

1. Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Corentin Travers. How to Solve Consensus in the Smallest Window of Synchrony. In *Proceedings of the 22nd International Symposium on Distributed Computing (DISC 2008)*, pages 32-46, 2008.

2. Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Corentin Travers. Of Choices, Failures and Asynchrony: The Many Faces of Set Agreement. In *Proceedings of the 20th International Symposium on Algorithms and Computation (ISAAC 2009)*, pages 943-953, 2009.

3. Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Morteza Zadimoghaddam. How Efficient Can Gossip Be? (On the Cost of Resilient Information Exchange). In *Proceedings of the 37th International Colloquium on Automata, Languages and Programming (ICALP 2010)*, pages 115-126, 2010.

4. Dan Alistarh, Seth Gilbert, Rachid Guerraoui, Zarko Milosevic, and Calvin Newport. Securing Every Bit: Authenticated Broadcast in Radio Networks. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2010)*, pages 50-59, 2010.

5. Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Corentin Travers. Generating Fast Indulgent Algorithms. In *Proceedings of the 12th International Conference on Distributed Computing and Networking (ICDCN 2011)*, pages 41-52, 2011.

6. Dan Alistarh, and James Aspnes. Sub-logarithmic Test-and-set Against a Weak Adversary. In *Proceedings of the 25th International Symposium on Distributed Computing (DISC 2011)*, pages 97-109, 2011.

7. Dan Alistarh, Hagit Attiya, Rachid Guerraoui, and Corentin Travers. Early Deciding Synchronous Renaming in $O(\log f)$ Rounds or Less. In *Proceedings of the 19th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2012)*, 2012.

8. Dan Alistarh, Rachid Guerraoui, Petr Kuznetsov, and Giuliano Losa. On the Cost of Composing Shared-Memory Algorithms. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2012)*, 2012.

9. Dan Alistarh, Michael Bender, Seth Gilbert, Rachid Guerraoui. How to Allocate Tasks Asynchronously. In *Proceedings of the 53rd IEEE Annual Symposium on Foundations of Computer Science (FOCS 2012)*, to appear, 2012.

10. Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Corentin Travers. Generating Fast Indulgent Algorithms. In the *Theory of Computer Systems Journal*, to appear, 2012.

11. Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Corentin Travers. Of Choices, Failures and Asynchrony: The Many Faces of Set Agreement. In *Algorithmica*, 62 (1-2), pages 595-629, 2012.

# Acknowledgements

To a great extent, the best part of my PhD has been getting to know and work with so many amazing people. Their ideas and their enthusiasm have made this a truly great experience.
First, I would like to thank my advisor, Prof. Rachid Guerraoui, for his continuing optimism, endless patience, and excellent advice throughout my PhD work, without which this thesis would not have been written.
Many thanks to Dr. Petr Kuznetsov, Prof. Paul Spirakis, and Prof. Ruediger Urbanke for carefully reviewing my thesis, and for providing me with invaluable comments, as well as to Prof. André Schiper for presiding the thesis jury.
I would also like to thank the people I have worked with over the years, in particular Prof. James Aspnes, Prof. Hagit Attiya, Prof. Michael Bender, Prof. Keren Censor-Hillel, Prof. Seth Gilbert, Dr. Petr Kuznetsov, Giuliano Losa, Prof. Corentin Travers, and Morteza Zadimoghaddam, for their great ideas, and unrelenting enthusiasm, without which this thesis would have been a lot less interesting. A special thanks goes to Prof. Seth Gilbert, whose encouragement and good humor have been invaluable during my early steps as a PhD student.
The current members of the Distributed Programming Lab, Radu Banabic, Victor Bushkov, Dr. Florian Huc, Mihai Letia, Giuliano Losa, Fabien Salvi, Vasileios Trigonakis, David Tudor, as well as the old guard, in particular Dr. Aleksandar Dragojevic, Dr. Vincent Gramoli, Dr. Nikola Knezevic, Dr. Michal Kapalka, Dr. Petr Kuznetsov, Dr. Maxime Monod, and Dr. Marko Vukolic, provided a great environment both on- and off-work. I am particularly indebted to Aleksandar, Nikola, and Petr, with whom I have had many interesting and stimulating discussions. Extra special thanks goes to Kristine Verhamme, for her great care, tremendous skills, and infinite patience in dealing with many crucial details during my stay at LPD.
I would like to thank my parents, Maria and Mihail, and my brother Alexandru for their love and support over the years. Special thanks goes to my friends Andrei, Dan Titus, Claudiu, Mihai, Cornelia&Dorin, Corina&Cosmin, Ana&Ion, Luisa&Lex, George, Rãzvan, and many others, for many good times together. Last but certainly not least, I would like to thank my ray of sunshine, Ludmila, for her love, encouragement, and insight throughout this time.

# Contents

# List of Figures

# 1 Introduction

Over the last forty years, computing power has continued to increase at an exponential pace. Often cited as “Moore’s Law", after Intel co-founder Gordon E. Moore, who described an instance of this trend in a 1965 article [74], this rule of thumb applies to several characteristics of digital electronic devices, such as processing speed, memory capacity, or even number and size of pixels in digital cameras [76].

Perhaps the most well-known instantiation of Moore’s law is in the context of the number of transistors in mass-produced microprocessors. Following this trend, computing power has continued to increase significantly over the last decades, while maintaining an accessible cost. For a long period, the increasing number of transistors on a chip has been correlated with a proportional increase in clock speed, which translated to a roughly exponential rate of increase in the single-thread performance of microprocessors. See Figure 1.1 for an illustration of this trend. From the software design perspective, this implied that a program would run faster on new machines simply because of increased clock speed, a phenomenon known as the software developer’s “free lunch" [85].

However, around 2004, due to technological limitations, processor manufacturers were forced to stop increasing processor clock rates. Instead, they started offering more *cores*, or processing units, as part of a single processor. (This trend is also depicted in Figure 1.1.) This change, known as the “multicore revolution," implies that software applications need to be designed with parallelism in mind, in order to exploit the power of the extra processing units in new processors. In general, parallel and concurrent programming, which had previously been considered the realm of specialized, high-performance computing, now needs to be taken into account in the design of most software applications.

The multi-core revolution is part of a more general trend in computing towards increased distribution. Indeed, over the last two decades, nearly all large-scale systems and services have become distributed, as they have evolved towards Internet-based or data-center-based architectures. This is the case for critical computing services, such as Internet search, information storage, or cloud computing. While at the processor level distribution is a consequence of

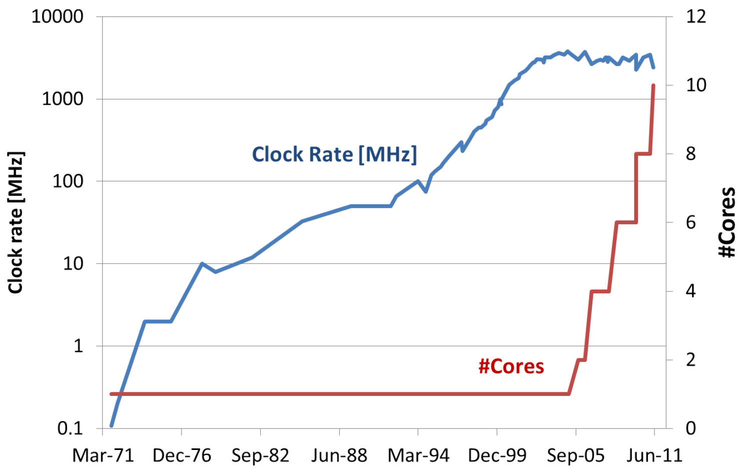

Figure 1.1: Overview of the evolution of clock rate versus number of cores in mass-produced microprocessors over the past forty years.

technological limitations, at the system level this trend is justified by the increased need for fault-tolerance and user locality. Another strong argument in favor of distribution, which has been noted both for processors and large-scale systems, is energy efficiency, e.g. [84, 11].

## 1.1 Distributed Computing

Distributing computation across multiple agents, such as servers or processor cores, introduces new challenges when compared to sequential, centralized computing. On the one hand, providing a distributed service or implementing a concurrent data structure poses new problems in terms of *availability*, or *liveness*, i.e. in terms of the progress guarantees that the implemented object can provide to its users. A second key challenge is maintaining the *consistency*, or *safety* of the object, whose implementation is distributed over multiple locations. The relation between availability and consistency for an implementation is most often expressed as a trade-off: by strengthening availability, the designer might have to weaken the consistency guarantees, and vice-versa.

Interestingly, there exists a set of core problems that keep recurring in distributed systems, irrespective of their architecture. These are problems such as agreement [67, 80], information dissemination [62], counting [12], and renaming [17]. Distributed (or concurrent) computing research has focused on these problems, since they often illustrate the computational power of a distributed system with respect to safety-liveness trade-offs, and the complexity of implementing basic primitives given the system's semantics.

The classic example of such a fundamental problem is *consensus* [67, 80], or *agreement*, a task in which a set of agents, or *processes*, needs to agree on a single decision among a set of proposed values. In particular, the *safety* requirements of consensus are that 1) the values returned by the processes need to be the same and 2) the value returned has to be a value proposed by some process; the *liveness* requirement is that each non-faulty process should eventually return a value.

Despite its apparent simplicity, the consensus task has been shown to be a fundamental abstraction in distributed systems: once consensus can be solved, any sequential object can be implemented on top of it via a *universal construction* [56]; consensus is also a key component for state-machine replication of distributed services, e.g. [30, 77]. On the other hand, Fischer, Lynch, and Paterson [49] showed that consensus cannot be implemented guaranteeing both liveness and safety in an asynchronous distributed system in which processes may crash, which is the standard model for large-scale message-passing or shared-memory systems. This impossibility, abbreviated as FLP, is probably the most celebrated result in distributed computing.

The FLP impossibility [49] illustrates a common occurrence in distributed computing, where several fundamental problems are either unsolvable or can be shown to have prohibitive complexity under asynchrony and crashes–literally hundreds of such impossibility results exist [47]. Since these fundamental problems can be seen as benchmarks for what can be implemented in a distributed system, a significant amount of research went into designing ways of circumventing such impossibilities.

One successful approach for avoiding the worst-case executions presented by the impossibility results has been *randomization*. In particular, for the consensus problem, for which FLP shows the existence of a schedule of infinite length as long as safety is preserved, it has been shown that such worst-case schedules can be avoided with probability 1 [28]. This does not rule out executions in which a process does not terminate, but guarantees that the probability of such executions is 0. It is important to note that the *safety* requirements of the problem are generally never broken by randomized algorithms, as they continue to hold in *every* execution.

## 1.2 Asynchronous Shared Memory

In this thesis, we focus on distributed computation in the standard *asynchronous shared-memory model* [70, 26]. In the context of shared memory, distributed algorithms are also called *concurrent*. In brief, in shared memory, $n$ processes communicate by reading and writing to shared registers. The processes are *asynchronous*, i.e. may execute at different speeds; in particular, there is no bound on their relative speeds. We assume that their timing is controlled by an adversarial *scheduler*, also known as the *adversary*. Processes may *crash* during the execution, in which case they stop taking steps as part of their algorithm; process crashes are also controlled by the adversary.

Some of the algorithms we present and analyze are *randomized*, in that processes may adapt their steps following the results of (local) random coin flips. We consider a *strong* adversary, which observes the state of processes and the results of random coin flips when deciding the execution schedule or crash pattern.

The asynchronous shared-memory model is the standard model for analyzing executions of concurrent data structures. Asynchrony and crashes model the fact that processes may be pre-empted or killed by the operating system scheduler during an execution, generating arbitrary interleavings of process steps. In this setting, we study algorithms that guarantee that each process terminates in a finite number of steps, known as *wait-free* algorithms. A precise definition of this model is given in Chapter 2.

## 1.3 The Renaming Problem

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

Formally, the renaming problem assumes that processes have unique initial names from a large, virtually unbounded namespace, and requires each process to eventually return a name (the *termination* condition), and that the names returned should be *unique* (the *uniqueness* condition). The size of the resulting namespace should be at most $M > 0$, which is given in advance. The namespace size $M$ should only depend on $n$, the maximum number of participating processes. The *adaptive* version of the renaming problem 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 size of the namespace matches exactly the number of participating processes, renaming is said to be *strong*, and the namespace is said to be *tight*. Otherwise, renaming is *loose*. Intuitively, a tight namespace is desirable since it minimizes the number of “wasted” names, which are allocated but go unused; in later chapters, we will see that *strong* renaming algorithms can in fact be used to implement other objects, such as counters and mutual exclusion.

A significant amount of research, e.g. [17, 27, 33, 72, 57, 4, 21, 44, 79], 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 [57, 34]. In fact, $(n + t - 1)$ is the best achievable namespace size when $t$ processes may crash [34, 35]. This result is an analogue of FLP for renaming, and its proof requires the use of complex topological techniques [57]. 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. [44].

## 1.4 Overview of the Results

In this thesis, we analyze the complexity of concurrent data structures in asynchronous shared memory. In particular, we focus on the complexity of *renaming*, as a benchmark problem for distributed coordination tasks. Despite considerable research effort on efficient algorithms for renaming, e.g. [79, 31, 72, 73, 4, 21, 44, 39, 2], prior to our work there have been no optimality results for shared-memory renaming, either for randomized or deterministic algorithms.

We present the first tight bounds for this problem, both for deterministic and randomized implementations. For deterministic algorithms, we give a tight *linear* lower bound on renaming into any sub-exponential namespace[1]. For randomized algorithms, we give the first tight upper and lower bounds on the time complexity of adaptive renaming, which are *logarithmic*. Together, our results give an exponential time complexity separation between deterministic and randomized implementations of this problem.

We also investigate connections between renaming and other shared objects. Since renaming can be solved trivially using objects with stronger semantics, such as stacks, queues, or fetch-and-increment counters, our lower bounds also apply to these widely-used objects. These results improve or match the previously known lower bounds for these problems (see Table 1.2 for an overview). On the other hand, our renaming algorithms can be extended to obtain new efficient implementations of other shared objects, such as counters, mutex, test-and-set or fetch-and-increment objects.

From a technical perspective, this thesis highlights new connections between renaming and other fundamental objects: sorting networks [64] and mutual exclusion [42]. We show that sorting networks can be used to obtain optimal-time solutions for randomized renaming. Such renaming solutions can also be used to obtain efficient mutual exclusion algorithms. We then proceed by reduction, and derive a lower bound on renaming from a known lower bound on the time complexity of mutual exclusion [63]. This result then generalizes to more complex objects that solve renaming. The lower bound on the time complexity of randomized renaming follows from a separate information-based argument.

Our results suggest that *deterministic* implementations of data structures solving renaming have worst-case schedules with *linear* time complexity. In essence, this implies that deterministic versions of widely-used data structures, e.g. queues, stacks, or counters, do not scale in the worst case, so a key practical question is how to circumvent this lower bound[2].

A natural alternative, which we emphasize in this thesis, is the use of *randomization*. We show that, using randomization, this lower bound can be circumvented for renaming, as we exhibit randomized algorithms with logarithmic step complexity, with high probability, i.e. exponentially better than allowed by the deterministic lower bound. Thus, a very tempting open question is whether our randomized techniques can be extended to obtain fast sub-

[1]This bound is matched by previously known algorithms. e.g. [72, 73]. See Section 7.6 for a detailed discussion.
[2]We discuss possible ways of circumventing the lower bound in Section 7.5.

linear versions of practical concurrent data structures. On the other hand, the logarithmic lower bound on the step complexity of randomized implementations suggests that there are complexity thresholds which cannot be avoided even with the use of randomization.

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

## 1.5 Contributions

### 1.5.1 The Time Complexity of Deterministic Renaming

The main contribution of this thesis is characterizing the time complexity of the renaming problem in asynchronous shared memory. For deterministic algorithms, we prove that $\Theta(k)$ process steps is a tight bound for the individual step complexity of adaptive renaming in a sub-exponential namespace[3] in the number of participants $k$. This result, whose proof can be found in Chapter 7, extends to non-adaptive renaming, to yield a linear lower bound on the complexity of non-adaptive renaming in a polynomial namespace in $n$. This is the first lower bound on the complexity of renaming in shared memory. It holds for wait-free algorithms using reads, writes, test-and-set, and compare-and-swap operations, and is matched by various algorithms in the literature, e.g. [72, 73] (for details, please see Table 1.2).

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

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

### 1.5.2 The Time Complexity of Randomized Adaptive Renaming

In Chapter 8 we complement the deterministic lower bound by also analyzing the time complexity of *randomized* adaptive renaming. More precisely, we analyze the worst-case expected *total* number of steps that processes must perform. We prove a global step complexity lower bound: given any algorithm that renames into a namespace of size $ck$, with $c \geq 1$, there exists an adversarial strategy that causes the $k$ processes to take $\Omega(k \log(k/c))$ total steps in expectation. This lower bound applies to algorithms using reads, writes, test-and-set and

[3]More precisely, a sub-exponential namespace is a namespace of size $o(e^k)$.

compare-and-swap primitives, and to algorithms that may not terminate with some non-zero probability. This total step complexity lower bound is tight for $c = 1$, i.e. for strong adaptive renaming, since it is matched by the renaming network algorithm presented in Chapter 6.

The same technique implies the same total step complexity lower bound for randomized implementations of approximate shared counters, i.e. counters that return a result that is within a factor of $c$ of the real value. The lower bound is tight within logarithmic factors for counters [12]. Since the lower bounds apply to randomized algorithms, and the almost-matching algorithm [12] is deterministic, this suggests that, against a strong adversary, the complexity improvement that may be obtained through randomization can be at most logarithmic. The lower bound also limits the complexity gain from allowing approximation within constant factors.

Our argument follows the structure of a previous result by Jayanti [60], which in turn is similar to a lower bound by Cook, Dwork, and Reischuk [40] on the complexity of computing basic logical operations on PRAM machines. Jayanti proved an $\Omega(\log k)$ lower bound on the expected step complexity of shared counters, queues, and stacks, which also applies to renaming. We generalize his result in two ways: first, we consider *total* step complexity, and thus obtain a stronger $\Omega(\log k)$ lower bound on the *average* worst-case expected step complexity of the problem. Second, our results also apply to loose (approximate) versions of renaming and counting, bounding the benefits of relaxing the object semantics.

### 1.5.3 Lower Bounds for Other Objects

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

In particular, the deterministic lower bound implies that wait-free deterministic implementations of these objects have linear step complexity in the worst case, suggesting that deterministic versions of these objects do not scale well in terms of worst-case time complexity. The bound holds for algorithms that are adaptive (and thus, have no bound on the number of processes $n$ that may access them in an execution), or if the algorithms do not assume any bound on the size of the initial namespace of participating processes. (For details on circumventing the lower bound, see Sections 7.4 and 7.5.) Thus, this linear lower bound could be circumvented by algorithms using randomization (as is the case for renaming), or by algorithms that assume processes already have names from a small space.

Similarly, the total step complexity lower bound also applies to queues, stacks, and fetch-and-increment out of read-write registers with compare-and-swap operations, giving an $\Omega(k \log k)$ lower bound for exact implementations in executions where $k$ processes participate. This bound is more general since it applies to randomized algorithms as well, and to algorithms

| Shared Object | Lower Bound | Type | Matching Algorithms | New Result |
|---|---|---|---|---|
| Deterministic $ck$-renaming | $\Omega(k)$ | Local | [73, 72, 21] | Yes |
| | $\Omega(k \log(k/c))$ | Global | - | Yes |
| Randomized $ck$-renaming | $\Omega(k \log(k/c))$ | Global | Chapter 6 | Yes |
| $c$-Approximate Counter | $\Omega(k \log(k/c))$ | Global | [12] | Yes |
| Fetch-and-Increment | $\Omega(k)$ | Local | [72] | Improves on [48] |
| | $\Omega(k \log k)$ | Global | Chapter 6 | Improves on [22] |
| Queues and Stacks | $\Omega(k)$ | Local | [56, 45] | Improves on [48] |
| | $\Omega(k \log k)$ | Global | - | Improves on [22] |

Figure 1.2: Summary of the lower bound results and relation to previous work.

that assume names from a small namespace. These lower bounds are matched by several known implementations (please see Table 1.2 for a case-by-case description).

### 1.5.4 Algorithms for Strong Randomized Renaming

The key algorithmic of contribution of this thesis is an algorithm for strong adaptive renaming based on a variation of sorting networks [41] which we call *renaming networks*. A renaming network is a sorting network in which all comparators have been replaced with two-process test-and-set objects. (See Section 6.1.1 for a brief introduction to sorting networks.)

The mechanism behind the algorithm is that each process is assigned a distinct input port, and follows a path through the network determined by leaving each comparator on its top output wire if it wins the test-and-set, and on the bottom output wire otherwise; the output name is the index of the port that it reaches. This construction guarantees that if $k$ processes enter the network on distinct input ports, they will reach the first $k$ output ports, thus returning unique names from 1 to $k$. The expected step complexity of the algorithm is bounded by the maximum number of comparators between an input port and an output port in the sorting network. There exist sorting networks for which this number is logarithmic in the number of input ports [5].

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

The resulting algorithm guarantees a tight adaptive namespace with complexity $O(\log k)$, with high probability. The construction and the proofs can be found in Sections 6.2 and 6.3.2.

This is the first known algorithm to achieve tight adaptive renaming in less than linear time.

It improves exponentially on previous strong renaming solutions, which had worst-case complexity at least linear, e.g. [9]. It also gives an exponential separation between deterministic and randomized renaming algorithms. The total work lower bound in Chapter 8 shows that this algorithm is in fact optimal, and that no asymptotic complexity improvements are possible by relaxing namespace size within constant factors. We build on this algorithm to obtain fast counters and bounded-use fetch-and-increment objects.

We complement this adaptive algorithm with a poly-logarithmic non-adaptive strong renaming solution that uses linear space, assuming hardware test-and-set operations. The algorithm, called *BitBatching*, assigns unique names to processes by repeatedly performing test-and-set operations over batches of bits of decreasing size. More precisely, we split a sequence of $n$ registers in batches of exponentially decreasing size, such that the first batch contains the first half of the registers, the second contains the next quarter, and so on, until we reach batches of size $\Theta(\log n)$. Processes perform test-and-set operations on $\Theta(\log n)$ registers in each batch, until first winning a test-and-set. A careful analysis shows that, somewhat surprisingly, every process obtains a test-and-set object before completing its test-and-set operations on the last batch, with high probability. Thus, the algorithm achieves strong renaming in $O(\log^2 n)$ steps with high probability, with $O(n \log n)$ total test-and-set operations.

### 1.5.5 Adaptive Randomized Test-and-Set

In Chapter 5, we present a randomized implementation of a test-and-set object with $O(\log k)$ step complexity. Intuitively, a test-and-set object works as a single-shot tournament for the $k$ participating processes: of the $k$ processes that call the test-and-set operation, a single process returns 0 (winner), while all the other processes return 1 (loser).

The algorithm, called *RatRace*, is based on a binary tree structure, which the processes first navigate from the root towards the leaves, in order to obtain a starting point in the tournament. Then, processes proceed towards the root, competing in two-process test-and-set objects along the way. The winner at the root is the winner of the test-and-set operation, and returns 0. Any other process loses and returns 1. The algorithm is *adaptive*, in that its complexity adapts to the number of processes $k$ participating in the execution. The construction is an adaptive version of the randomized test-and-set first proposed by Afek et al. [3]. The algorithm can be modified to obtain a fast solution for loose randomized renaming, which comes in useful as part of the adaptive renaming algorithm based on sorting networks, presented in Chapter 6.

## 1.6 Roadmap

The thesis is divided in three parts. (Note that the technical presentation has a slightly different structure than the introduction.) The first part presents some background on shared-memory distributed computing, presenting the model, problem statements, and an overview of related work. In turn, each later chapter contains precise definitions of the objects used in the chapter, and a precise comparison of the results with previous work.

Part II presents the algorithms described in the introduction. In particular, Chapter 5 presents the adaptive test-and-set algorithm and the resulting loose renaming algorithm. Chapter 6 presents the strong adaptive renaming algorithm based on sorting networks, its applications to counting objects, and the non-adaptive *BitBatching* algorithm.

We focus on lower bounds in Part III. We give the deterministic lower bound and its extensions to other objects in Chapter 7. We then present the proof of the global step complexity lower bound in Chapter 8, with its applications to renaming and counting. Both chapters also contain overviews of possible ways of circumventing the lower bounds. We summarize the results and present an overview of open questions in Chapter 9.

# Preliminaries Part I

# 2 System Model

In this chapter, we introduce the system model for which our algorithms and lower bounds are designed. We also describe the cost measures under which we will analyze algorithms. In brief, the model we consider is the classical *asynchronous shared memory* model [26, 70] in which processes execute without bounds on their relative execution speed, in the presence of crash failures, communicating through operations on registers.

## 2.1 The Asynchronous Shared Memory Model

### 2.1.1 Model Overview

We consider the standard asynchronous shared-memory model, in which $n$ processes $p_1, \ldots, p_n$ communicate through operations on shared multi-writer multi-reader atomic registers [26, 70]. We will denote by $k$ the *contention* in an execution, i.e. the actual number of processes that take steps in the execution.

Processes follow and algorithm, which is composed of *steps*. Each step 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. A number of $t < n$ processes may fail by crashing. A *failed* process does not take any further steps. A process that does not crash during an execution is called *correct*.

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

In the following, we define these terms more formally.

### 2.1.2 Processes, Algorithms, and Shared Objects

A *process* is a sequential unit of computation, created by an application when necessary. For simplicity, we will assume that processes are created at the beginning of each execution, and each executes steps from its algorithm when scheduled. Thus, we denote by $\Pi = \{p_1, p_2, \ldots, p_n\}$ the set of all processes that may execute an algorithm. Accordingly, $n$ will be the total number of processes that may execute an algorithm. On the other hand, we will denote by $k$ the *actual* number of processes that take steps in the execution. Consequently, we have the relation $k \leq n$.

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

Each process executes an algorithm assigned to it by the application. The algorithms we analyze are either *deterministic*, where the process's next operation is always determined by its state, or *randomized*, in which case the process's next operation may be determined by its state, and the results of random coin flips performed by the process.

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

$$val \leftarrow R.read()$$

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

### 2.1.3 Progress Conditions

An algorithm is *wait-free* if it ensures that every call by a correct process returns within a finite number of steps [58]. A *lock-free* algorithm ensures that, given any schedule of infinite length, infinitely often *some* method call finishes in a finite number of steps. Clearly, any wait-free implementation is also lock-free, whereas the converse is not true, since lock-free algorithms may allow operations by correct processes that never return. An algorithm is *obstruction-free* if, from any point in an execution from which a process runs in isolation, the process terminates in a finite number of steps. A lock-free algorithm is obstruction-free, but not vice-versa.

### 2.1.4 Randomization

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

Processes may perform local coin flips by calling a local function *coin*, which takes two integer parameters $a$ and $b$ with $a \leq b$, and returns an integer $x$ with $a \leq x \leq b$ chosen uniformly at random. For example, the call

$$coin(0, 1)$$

will return 0 with probability 1/2, and 1 with probability 1/2.

### 2.1.5 Object Implementations

In this thesis, we will consider *implementations* of shared objects. An implementation $I_X$ of an object $X$ consists of $n$ algorithms, one for each process $p_1, \ldots, p_n$ that might execute the implementation. When a process $p_i$ invokes an operation *op* on object $X$ with implementation $I_X$, process $p_i$ will follow algorithm $I_X(i)$ until it receives a response from the operation *op*. In short, we say that $p_i$ executes $I_X$. Thus, a shared object may be seen as an abstraction to which we map a set of operations that a process takes in the execution.

Shared object implementations will be based on *base objects*, which are the primitives that form the shared memory, and on other lower level shared objects. Base objects are the primitives given by the shared memory, for example by hardware or the runtime environment. They are entities separate from the processes, and processes are not aware of their implementations. Unless otherwise specified, every process has access to every base object.

Second, a shared object implementation can also use other implementations of lower-level shared objects. (In turn, these may be composed of other base and shared objects, and so on.) Throughout this thesis, an *object* will be either a base object, or a shared object. If $Q$ is the set of base and shared objects used by implementation $I_X$ of $X$, we say that $X$ is *implemented from* the objects in $Q$.

In brief, an algorithm implementing an object $O$ at process $p_i$ can be seen as a set of sub-algorithms and base objects. Process $p_i$ follows the algorithm, executing either local computation steps, or operations on shared or base objects. For any implementation $I_X$ of a shared object $X$ used by the algorithm, process $p_i$ executes algorithm $I_X(i)$ for the implementation of $X$. In particular, process $p_i$ may execute operations on the base objects, which provide the means by which processes communicate with each other.

### 2.1.6 Concurrent Executions and the Adversarial Scheduler

An execution is a sequence of operations performed by a set of processes. In order to represent executions, we will assume discrete time, where at every unit there is only one active process. In a time unit, the active process can perform any number of local computations or local coin flips, and then issue an event or execute a *step*. A step is an execution of an operation on a base object, which comprises the invocation and the subsequent response (therefore, every operation on a base object takes at most one unit of time to execute). Whenever a process $p_i$ becomes active (as decided by the scheduler), $p_i$ executes an event or a step. It may be possible that a process does not have anything to execute, e.g. if it terminates its algorithm, in which case it executes an empty no-op step.

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

The order in which processes take steps and issue events is determined by an external abstraction called a *scheduler*, over which processes do not have control. In the following, we will consider the scheduler as an *adversary*, whose goal is to maximize the cost of the protocol (in this thesis, we focus on running time as a cost measure). Thus, we will use the terms adversary and scheduler interchangeably. The adversary controls the *schedule*, which is a (possibly infinite) sequence of process identifiers. If process $p_i$ is in position $t$ of the sequence, then this implies that $p_i$ is active at time $t$. The adversary has the freedom to schedule any interleaving that complies with the given model. In this thesis, 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 randomized algorithms, notice that different assumptions on the relation between the scheduler and the random coin flips that processes perform during an execution may lead to different results. In this thesis, we will assume that the adversary controlling the schedule is a *strong* adversary, that observes the results of the local coin flips, together with the state of all processes, before scheduling the next process step (in particular, the interleaving of process steps may depend on the result of their coin flips).

This is the standard adversarial model for randomized distributed algorithms, which reflects the fact that the speed of a process may be influenced by the results of random coin flips that the process performs. On the one hand, it is the strongest “reasonable" adversarial model, since a stronger adversary would have to be aware of the results of coin flips that the processes perform in the future. On the other hand, it encompasses weaker adversarial models, such as the *oblivious* adversary, e.g. [6], which fixes the scheduling and failure pattern independently of the processes' coin flips.

## 2.2 Asynchrony and Wait-Freedom

In this thesis, we focus on *asynchronous* shared-memory systems. In such systems, the time delay between two consecutive events of any process may be arbitrary, i.e. there are no assumptions on the relative speed of processes. This models the fact that, in general-purpose systems, processes may be preempted or otherwise delayed for arbitrary periods of time. While real-world systems may not be entirely asynchronous, proving algorithms correct in the asynchronous model ensures that they will be correct in any system in which delays are bounded.

An implementation is *wait-free* if any process invoking an operation also returns within some finite number of its own steps. More precisely, note that we are in a system where some number $t < n$ of processes may fail by crashing. A crashed process stops taking any steps in the remainder of the execution, and therefore it is scheduled for a finite number of times in the execution. On the other hand, there may be processes that are scheduled for an infinite number of times: these processes do not crash, and are called *correct*. An algorithm is said to be wait-free if, whenever a correct process $p_i$ invokes an operation on object $X$, $p_i$ returns from that operation.

## 2.3 Complexity Measures

We measure complexity in terms of process steps: each shared-memory operation is counted as one step–local computation, including coin flips, is not counted. 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. The *total* step complexity is the total number of shared memory operations that all participating processes perform during an execution. For randomized algorithms, we will analyze the worst-case *expected* number of steps that a process may perform during an execution as a consequence of the adversarial scheduler, or give more precise probability bounds for the number of steps performed during an execution.

For the lower bound in Chapter 7, we will use a stronger measure of complexity, by counting the number of *remote memory references* (RMRs). In cache-coherent (CC) shared memory, each process maintains local copies of shared variables inside its cache. The consistency of the cache among processes is ensured by a coherence protocol. A variable is *remote* to a process if its cache contains a copy of the variable whose value is out of date (or if the cache contains no copy of the variable); otherwise, the variable is *local*. A process step is *local* if it accesses a local variable. Otherwise, the step is a *remote memory reference* (RMR). A similar definition exists for the distributed shared memory (DSM) model. Notice that, since each RMR implies a distinct process step, RMR complexity is always a lower bound on step complexity.

## 2.4 Linearizability

Linearizability [59] is a correctness condition for shared object implementations. Intuitively, an implementation is linearizable (or atomic) if every shared memory operation should appear to the processes as if it was executed instantaneously at some single and unique point in time between its invocation and its response. This notion helps keep the algorithms simple, by eliminating technical details and providing a clean interface. The semantics of atomic objects can be described by using their sequential behavior, i.e. by giving their sequential specification.

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

**Linearizability and Randomization.** Recent results by Golab et al. [51] show that linearizability is not a sufficient correctness condition when randomization is employed. More precisely, they show that the adversary can gain extra power whenever a randomized algorithm uses other (deterministic or randomized) linearizable implementations as sub-algorithms. In this thesis, we circumvent this technical issue by avoiding the use of linearizability as a correctness condition when employing sub-algorithms: instead, we isolate a set of invariants whenever we use a known implementation as a sub-algorithm. (For example, this is the reason we isolate a set of specific properties for the test-and-set implementations in Section 5.2.1 instead of using their linearizability directly.)

## 2.5 Probabilistic Preliminaries

### 2.5.1 Basics

We give a brief overview of the classic definitions and properties of probability space, probability measure, independence, random variable, and expectation. We follow the presentation of standard texts on randomized algorithms, e.g. [75, 71].

**Definition 1.** *A* probability space *has three components:*

1. *A* sample space $\Omega$, *which is the set of all possible outcomes of the random process modeled by the probability space;*

2. *A family of sets $\mathcal{F}$ representing the allowable events, where each set in $\mathcal{F}$ is a subset of the sample space $\Omega$;*

3. *A probability function* $\Pr : \mathscr{F} \to \mathbb{R}$*, satisfying Definition 2.*

An element $E$ of $\Omega$ is called an *elementary event*.

**Definition 2.** *A* probability function *is any function* $\Pr : \mathscr{F} \to \mathbb{R}$ *satisfying the following conditions:*

1. *For any event* $E$*,* $0 \le \Pr(E) \le 1$*;*
2. $\Pr(\Omega) = 1$*;*
3. *For any finite or countably finite sequence of pairwise mutually disjoint events* $E_1, E_2, \ldots$

$$\Pr\left(\bigcup_{i\ge 1} E_i\right) = \sum_{i\ge 1} \Pr(E_i).$$

In the rest of this thesis, we will use *discrete* probability spaces, in which the sample space $\Omega$ is finite or countably infinite, and the family $\mathscr{F}$ of allowable events consists of all subsets of $\Omega$. In a discrete probability space, the probability function is uniquely defined by the probabilities of the elementary events.

A consequence of Definition 2 is known as the *union bound*, which will prove useful in the rest of the thesis.

**Proposition 1** (Union Bound)**.** *For any finite or countably infinite sequence of events* $E_1, E_2, \ldots$*,*

$$\Pr\left(\bigcup_{i\ge 1} E_i\right) \le \sum_{i\ge 1} \Pr(E_i).$$

When analyzing a random process, we are often interested in some value associated to the event rather than in the event itself. For this, we introduce the notion of *random variable*.

**Definition 3.** *A* random variable $X$ *on a sample space* $\Omega$ *is a real-valued function on* $\Omega$*; that is,* $X : \Omega \to \mathbb{R}$*, such that, for all* $x \in \mathbb{R}$*,*

$$\{E \in \Omega \mid X(E) \le x\} \in \mathscr{F}.$$

*Thus, we will denote by* $\Pr[X \le x]$ *the probability* $\Pr[\{E \in \Omega \mid X(E) \le x\}]$*. A* discrete *random variable is a random variable that may take only a finite or countably infinite number of values.*

We now define the notion of *independence* of events and of random variables.

**Definition 4.** *Events* $E_1, E_2, \ldots, E_\ell$ *are* mutually independent *if and only if, for any subset* $I \subseteq \{1, 2, \ldots, \ell\}$*,*

$$\Pr\left(\bigcap_{i\in I} E_i\right) = \prod_{i\in I} \Pr(E_i).$$

*Similarly, random variables $X_1, X_2, \ldots, X_\ell$ are* mutually independent *if and only if, for any subset $I \subseteq \{1,2,\ldots,\ell\}$, and any values $x_i$ with $i \in I$,*

$$\Pr\left(\bigcap_{i\in I}(X_i = x_i)\right) = \prod_{i\in I}\Pr(X_i = x_i).$$

A basic characteristic of a random variable is its *expectation*, which can be seen as a weighted average of the values it assumes, where each value is weighted over the probability that the variable assumes that value.

**Definition 5.** *The* expectation *of a discrete random variable $X$, denoted by $E[x]$, is given by*

$$E[X] = \sum_i i \cdot \Pr(X = i),$$

*where the sum is taken over all the values in the range of $X$. The expectation is finite if the sum converges, otherwise it is unbounded.*

**With high probability.** We say that an event occurs *with high probability* in a parameter $x$ if it occurs with probability at least

$$1 - O\left(\frac{1}{x^c}\right), \text{ for some constant } c \geq 2.$$

In general, we pick the parameter $x$ to be large, so that the probability that the event does not occur is practically negligible. Notice that the union of a finite number of events that occur with high probability in a parameter $x$ is also an event that occurs with high probability in $x$.

### 2.5.2 Distributions

In this context, we define the following distributions, which will appear throughout the thesis. We adopt the presentation of [75].

**Bernoulli distribution.** Suppose we flip a coin whose probability of Heads is $p$. Let $X$ be a random variable with value 1 when the result of the coin flip is Heads, and 0 otherwise. Then $X$ has the *Bernoulli* distribution with parameter $p$. Also, $E[X] = p$.

**Geometric distribution.** Suppose we flip a coin repeatedly until Heads appears for the first time. Assuming that each coin flip has the Bernoulli distribution with parameter $p$, the random variable $X$ denoting the total number of coin flips has the *geometric distribution* with parameter $p$. Then $E[X] = 1/p$.

**Negative Binomial Distribution.** Let $X_1, X_2, \ldots, X_n$ be independent identically distributed (i.i.d) random variables whose common distribution is the geometric distribution with parameter $p$. The random variable $X = X_1 + \ldots + X_n$ denotes the number of coin flips needed to obtain

$n$ Heads. The random variable $X$ has the negative binomial distribution with parameters $n$ and $p$. The density function for this distribution is defined only for $x = n, n+1, n+2, \ldots$:

$$Pr[X = x] = \binom{x-1}{n-1} p^n (1-p)^{x-n}.$$

The expected value of $X$ is $E[X] = n/p$.

### 2.5.3 Complexity Measures, Adversaries, and Randomization

In Section 2.3, we defined the complexity measures we focus on in this thesis, in particular individual and global step complexity. Given that some of the algorithms we consider are randomized, the processes' coin flips during the execution will induce probability distributions over the complexity of the algorithms. We now define step complexity in the context of randomization more precisely. We adopt the standard presentation of [26].

As seen in Section 2.1.6, we view the scheduler as an adversary whose goal is to extend the execution of the algorithm for as long as possible. Thus, the adversary can be seen as a function that takes an execution prefix as an argument, and returns the next process to be scheduled to take a step. The adversary must satisfy the admissibility conditions for the asynchronous shared-memory model. Since we assume a *strong* adversary, the execution prefix the adversary receives also contains the results of the random coin flips that processes performed in the execution.

Let $\mathscr{R}$ be the set of all possible outcomes of coin flips that processes may perform during an execution. Notice that an execution of an algorithm is uniquely determined by a series of random coin flips $R$ that the processes perform, and an adversary $D$. We denote this execution by $exec(R, D)$. Let $P$ be some predicate on an execution, for example the fact that each correct process completes within some number of steps. Given a particular adversary $D$, we define the probability of $P$ under $D$ as

$$\Pr_D[P] = \Pr[\{R \in \mathscr{R} \mid exec(R, D) \text{ satisfies } P\}].$$

Similarly, let $T$ be a random variable defined on an execution, for example the maximum number of steps that a process performs during the execution. Then we define the expectation of $T$ under an adversary $D$ as

$$E_D[T] = \sum_{x \text{ value of } T} x \cdot \Pr_D[T = x].$$

Since we are interested in the worst-case performance of algorithms, we will consider these complexity measures under all possible adversaries. Thus, if we denote by $T$ the random variable describing the step complexity of an algorithm $A$ in an execution, then the worst-case expected step complexity of $A$ is defined as

$$E[T] = \max_{\text{all adversaries } D} E_D[T].$$

Similarly, the probability of a predicate $P$ is considered over all adversaries $D$. Thus, we say that a predicate $P$ holds with probability at least $p$ if

$$\Pr_D[P] \geq p, \text{ for all adversaries } D.$$

For example, if we say that the step complexity of an algorithm is $O(\log n)$ with high probability in $n$, we mean that the probability that the number of steps a process takes during an execution of the algorithm exceeds $\Theta(\log n)$ is upper bounded by $O(1/n^c)$, for some constant $c \geq 2$, and for all adversaries $D$.

In general, given a shared-memory object $O$, we aim to obtain upper and lower bounds on the worst-case expected step complexity of its randomized implementations. In some cases, we are able to obtain stronger “with high probability" upper bounds on the complexity of the object. Such upper bounds are preferable since they give a better measure of the concentration of the random variable representing the complexity of the implementation.

# 3 Problem Statements

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

## 3.1 Renaming

The *renaming problem*, introduced in [17], is defined as follows. Each of the $n$ processes has initially a distinct identifier $id_i$ taken from an unbounded ordered domain, and should return an output name $m_i$. Given an integer $M$, an object ensuring *deterministic* renaming into a namespace of size $M$, also called an $M$*-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 $M$.
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.

Note that the domain of values returned, which we call the *target namespace*, is of size $M$. In the classical renaming problem [17], the parameter $M$ 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 should only depend on the number of participating processes $k$ in the current execution. In this instance of the problem, we assume that the processes do not know the maximum number of participating processes $n$, so the complexity of the protocol may only depend on the actual number of participants $k$.

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 $n$. 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 known, as the *non-adaptive* renaming problem.

## 3.2 Registers

The simplest base object we will use is the *register*. Every register supports two operations:

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

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

## 3.3 Test-and-Set and Compare-and-Swap

The *test-and-set* object, whose sequential specification is given in Figure 3.1, 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 process entering the object returns 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 $\mathscr{L}$ 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 all later responses are either *1* or the caller crashes.
4. (Uniqueness.) At most one process may return *0*.

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

Figure 3.1: 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 3.2: Sequential specification of the *compare-and-swap* object.

5. (Non-triviality.) If a process accesses the object in isolation and for the first time, then it returns *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.

The specification above describes a single-use (*one-shot*) test-and-set object, which can be employed only once (since subsequent calls will always return value 1). A stronger variant of the test-and-set in Figure 3.1 is a multi-use (*re-settable*) test-and-set, whose value can be set back to 0 by the process that last returned 0 from the object, i.e. by the current winner.

The *compare-and-swap* object can be seen a generalization of the test-and-set object, whose underlying register supports multiple values (as opposed to only 0 and 1). Its sequential specification is presented in Figure 3.2. 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.

## 3.4 Consensus

Another object that we will be referring to throughout this thesis is the *consensus* object. This object encapsulates the notion of agreement in a distributed setting, and is a key tool to understand the computational power of shared-memory abstractions [56]. Its definition is as follows.

Each process $p_i$ has an input value $v_i$, which, for simplicity, is assumed to be an integer. Each process should decide on an output value $y_i$. An algorithm for deterministic consensus satisfies the following properties:

- *Agreement*: For every correct processes $p_i$ and $p_j$, their corresponding output values are equal, i.e. $y_i = y_j$.
- *Validity*: For every correct process $p_i$, if $p_i$ decides $y_i$, then $y_i = v_j$ for some process $p_j$.
- *Termination*: In every execution, with probability 1, every correct process $p_i$ eventually decides a value $y_i$.

The *randomized consensus* problem replaces the termination condition with *randomized termination*: with probability 1, every correct process $p_i$ eventually decides a value $y_i$.

## 3.5 Counter Objects

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

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

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

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

# 4 Related Work

In this chapter, we provide a general overview of research on renaming and related data structures in the asynchronous shared memory model. Each of the later chapters contains a precise discussion of its results in relation to previous work.

## 4.1 Asynchronous Shared Memory

Historically, the asynchronous shared memory model arose in the study of early operating systems, in which several *processes* can run on a single processor, sharing memory, with possibly-arbitrary interleavings of steps. Currently, variants of the asynchronous shared memory model are used to analyze programs running on multi-processors, in which processes may run on separate processors and communicate through shared memory. (The model is defined in Section 2.1.)

Asynchronous shared-memory systems have been shown to have lots in common with asynchronous networks, since the two models present similar algorithms and impossibility results, even in the presence of faults. Moreover, there exist generic simulations between the two models [16], allowing algorithms designed for one model to be executed in the other model.

Some fundamental objects studied in the context of the asynchronous shared memory model are registers, mutual exclusion, consensus, and renaming. In particular, a significant amount of research has studied the constructibility of strong types of registers, such as atomic multi-writer multi-reader registers, from registers with weaker semantics. For an overview of shared-memory register transformations, we refer the reader to [58]. (In this thesis, we assume multi-writer multi-reader atomic registers.)

The mutual exclusion problem, defined formally in Section 7.1.1, was first identified and solved by Edsger W. Dijkstra in a seminal 1965 paper [42]. Subsequent research has studied the solvability and complexity of this problem in variations of the asynchronous shared-memory model, such as cache-coherent shared memory (CC), or distributed shared memory (DSM). We refer the reader to the surveys by Raynal [83] and Taubenfeld [86] for a detailed overview of this line of research.

The *consensus* problem, defined in Section 3.4, is arguably the most studied problem in distributed computing. A fundamental result by Fischer, Lynch, and Patterson [49] showed that consensus is impossible in an asynchronous system in which one process may crash. The original result was stated for asynchronous networks, and was adapted to asynchronous shared memory by Loui and Abu-amara [69]. Significant research effort went into circumventing this impossibility, either by using stronger timing assumptions, e.g. [43, 20], by using failure detectors, which encapsulate the synchrony assumptions needed to overcome these impossibilities, e.g. [37, 36], or by the use of randomization, e.g. [15, 18]. In particular, it is known that $\Theta(n^2)$ expected total process steps are necessary and sufficient to achieve asynchronous shared-memory consensus [18].

## 4.2 Renaming

The renaming problem, defined in Section 3.1, can be seen as the dual of the consensus problem: if for consensus processes have to *agree* on a single value, in the renaming problem processes have to *disagree*, i.e. return distinct values from a small namespace. The problem was introduced by Attiya et al. [17], in the asynchronous network model. The paper presented a non-adaptive algorithm that achieves $(2n-1)$ names in the presence of $t < n$ 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.

For synchronous message-passing systems, Chaudhuri et al. [38] 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. 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 [78] 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 [46]. Recently, Alistarh et al. [10] 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.

The first shared-memory renaming algorithm was given by Bar-Noy and Dolev [27], 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 [33] proved that long-lived renaming in a namespace of size $m(k)$ is impossible in asynchronous shared memory using reads and writes if $m(k) < 2k-1$. They also gave the first long-lived $(2k-1)$-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 [57] used algebraic topology to show that there exist values of $k$ for which $(2k-2)$-renaming is impossible. Recently, Castañeda and Rajsbaum [34, 35] proved that if $k$ is a prime power, then $m(k) \geq 2k-1$ is necessary, and, otherwise, there exists an algorithm with $m(k) = 2k-2$. *Anonymous* renaming, where processes do not have initial identifiers, 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. A formal version of this argument can be found in [68, 65].

Afek and Merritt [4] presented an adaptive read-write renaming algorithm with optimal namespace of size $(2k-1)$, and $O(k^2)$ step complexity. Attiya and Fouren [21] gave an adaptive $(6k-1)$-renaming algorithm with $O(k \log k)$ step complexity. Chlebus and Kowalski [39] gave an adaptive $(8k - \log k - 1)$-renaming algorithm with $O(k)$ step complexity. For *long-lived* adaptive renaming, there exist implementations with $O(k^2)$ time complexity for renaming into a namespace of size $O(k^2)$, e.g. [2]. The fastest such algorithm with optimal $(2k-1)$ namespace size has $O(k^4)$ step complexity [21]. Using load-linked and store-conditional primitives, Brodsky et al. [32] gave a linear-time algorithm with a tight namespace. (Their paper also presents an efficient synchronous shared-memory algorithm.)

The relation between renaming and stronger primitives such as fetch-and-increment or test-and-set was investigated by Moir and Anderson [72]. 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. (An efficient generalization of this strategy is given in Section 6.5.) Moir and Anderson [72] also present implementations of renaming from registers supporting set-first-zero and bitwise-and operations.

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, 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. [79]. 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. [44]. The authors obtain a *strong* non-adaptive renaming algorithm based on the randomized wait-free test-and-set implementation of Afek et al. [3]. Their algorithm is long-lived, and is shown to have amortized step complexity $O(n \log n)$. The average-case total step complexity is $\Theta(n^3)$.

## 4.3 Counting Data Structures

Many multi-processor coordination tasks can be expressed as counting problems, where processes assign values from a given range. Thus, there has been considerable work on counting data structures over the last few decades. *Counting networks* [14] are an example of such data structures, where processes interact with a counter by traversing a network of *balancer* objects[1]. Efficient counting networks with $O(\text{ polylog } n\ )$ complexity are known [14]. Counting networks are similar to the renaming networks presented in Section 6.2.1: however, the aim of a counting network is to balance the number of processes exiting on the output ports, whereas renaming networks ensure that no two processes reach the same output port. As a consequence, the structure and applications of counting networks are in general different than those of renaming networks.

Another well-studied object is the *counter*. A linear-time deterministic atomic counter implementation follows from the atomic snapshot construction of Afek et al. [1]. Jayanti et al. [61] proved $\Omega(n)$ space and time lower bounds for deterministic counter implementations if processes may access the object multiple times, while Jayanti [60] gave $\Omega(\log n)$ time lower bounds for counter objects using reads, writes, or load-linked/store-conditional operations.

A technical breakthrough by Aspnes et al. [12] leveraged the fact that objects may be accessed for a limited number of times to give an $m$-valued max register implementation with $O(\log m)$ time complexity, and a $O(\log m \log n)$ upper bound for deterministic wait-free counters with maximal value $m$. Recently [13], this technique was generalized to obtain atomic snapshots with $O(\log^2 b \cdot \log n)$ time complexity for a scan, and $O(\log b)$ complexity for an update, where $b$ is the number of update operations performed on the object.

[1]Intuitively, a balancer acts as a *toggle* mechanism. It has two inputs and two outputs; processes enter the object on its inputs, and the balancer alternates sending processes to its top and bottom output wires.

# Algorithms Part II

# 5 Test-and-Set in $O(\log k)$ Steps

In this chapter, we present randomized algorithms for adaptive test-and-set and adaptive renaming. We begin by presenting an adaptive test-and-set implementation with step complexity $O(\log k)$ in executions where $k$ processes participate, with high probability. We then notice that this implementation can be used to solve renaming into a polynomial namespace in $k$, with $O(\log k)$ complexity, both with high probability.

We start by presenting an adaptive implementation of a randomized adaptive test-and-set object. The implementation we present is *single-use*, or *one-shot*; however, it can be easily transformed into a multi-use implementation using standard techniques, e.g. [3]. The definition and sequential specification of the *test-and-set* object are given in Section 3.3. In brief, a test-and-set object can be seen as a tournament object for $n$ processes. 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 test-and-set by returning the initial value 0, while all other processes *lose* the test-and-set object, returning 1. A key property is that no losing test-and-set operation may return before the winning operation is invoked.

Note that one-shot test-and-set cannot be implemented deterministically wait-free in asynchronous shared memory, since it can solve *consensus* for two processes; see [56]. We present an efficient randomized implementation that guarantees the desired properties with probability 1, and is linearizable, following the definition given in [59]. Our implementation is *adaptive*, in that the complexity of an operation depends on the contention $k$ at the object, and not on $n$, the total number of processes. An implementation of test-and-set that ensures the linearization property will provide a total order over the completed *test-and-set* invocations, so that the resulting sequence of operations is consistent with a sequential execution.

We begin the presentation by describing some auxiliary objects used by our algorithm. In Section 5.2, we describe the implementation in detail, and analyze its correctness and complexity. In Section 5.3, we show that the algorithm can be easily modified to obtain an adaptive renaming solution. We give an overview of related work in Section 5.4.

```
Shared:
$G, S$, atomic registers, initially $\bot$
procedure split($id_i$)
    $G \leftarrow id_i$
    if $S$ = true then
        return right
    $S \leftarrow$ true
    if $G = id_i$ then
        return stop
    else
        return left
```

Figure 5.1: Implementation of a *deterministic* splitter object.

```
Shared:
$G, S$, atomic registers, initially $\bot$
procedure split($id_i$)
    $G \leftarrow id_i$
    if $S$ = true then
        if coin(0,1) = 0 then return right
        else return left
    $S \leftarrow$ true
    if $G = id_i$ then return stop
    else
        if coin(0,1) = 0 then return right
        else return left
```

Figure 5.2: Implementation of a *randomized* splitter object.

## 5.1 Auxiliary Objects

### 5.1.1 Splitters and Randomized Splitters

A *splitter* object [66, 72] is a weak synchronization primitive for $n$ processes. It provides a single operation *split*, which processes call with their initial identifier $id_i$ as an argument. Each correct process should return one of three values {*left*, *right*, *stop*}. The object guarantees the following properties.

- 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*.
- At most one process returns *stop*.

Notice that the first property implies that if a single correct process calls *split*, then the process returns *stop*. The splitter object can be implemented wait-free using registers [66, 72]. The pseudocode for the implementation is given in Figure 5.3.

The *randomized* splitter object is a variant of the splitter providing probabilistic guarantees on the number of processes returning *left* or *right*. It ensures 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.

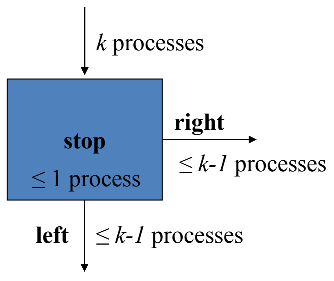

Figure 5.3: Deterministic splitter.

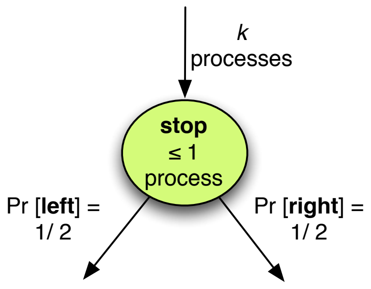

Figure 5.4: Randomized splitter.

The randomized splitter was introduced in [25], where it was shown that it can be implemented wait-free using registers. The pseudocode for the implementation is given in Figure 5.4.

**Splitters and Renaming.** The splitter object was introduced in the context of mutual exclusion [66]. Another interesting use of this object is in the context of the renaming problem. Anderson and Moir [72] noticed that splitters can be connected in a rectangular grid, as described in Figure 5.7. Since 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 labeling of the splitters as in Figure 5.7, 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 was introduced in the context of the adaptive collect problem [25]. The authors noticed that a binary tree of randomized splitter objects can be used to assign unique identifiers to processes from a namespace which only depends on the number of participants $k$. We analyze the properties of this procedure in the next section.

### 5.1.2 Randomized Test-and-Set for Two and Three Processes

Our test-and-set implementation for $n$ processes uses a randomized two-process test-and-set implementation using only registers. This object satisfies the definition of randomized test-and-set as given in Section 3.3, assuming that only two processes may access the object. The implementation we use is that of Tromp and Vitanyi [87], described in Figure 5.5.

**Description.** The algorithm uses two four-valued registers $R_0$ and $R_1$. The four possible values of these registers are *me*, *he*, *choose*, and *rst*. Each process $p_i$ solely writes to register $R_i$, and only reads the variable $R_{1-i}$. Each process first checks whether the object has already been won. If not, then the process proposes itself as the winner, and proceeds to perform a sequence of asynchronous choosing rounds. In each such round, the process marks its register with value *choose*, then it reads the value of the other process's register. If the other process gave up, i.e. $R_{1-i} = he$, or the other processes is choosing as well *and* the local coin flip returns 0, then the process proposes itself as the winner. Otherwise, the process gives up by writing *he* to the

```
**Shared**:
Registers $R_0, R_1$
**procedure** *test-and-set*()
  **if** $R_i$ = he *and* $R_{1-i} \neq$ rst **then**
      **return** 1
  $R_i \leftarrow me$
  **while** $R_{1-i} = R_i$ **do**
      $R_i \leftarrow choose$
      **if** $R_{1-i}$ = he *or* ($R_{1-i}$ = choose *and* coin(0,1) = 0) **then**
          $R_i \leftarrow me$
      **else** $R_i \leftarrow he$
  **if** $R_i$ = me **then** **return** 0
  **else** **return** 1
```

Figure 5.5: Implementation of the two-process test-and-set object [87].

register $R_i$. The algorithm returns as soon as the two processes have written distinct values in their registers. If the register $R_i$ has value *me*, then the process is the winner of the current instance of the *test-and-set* operation; otherwise, the process loses and returns 1.

Notice that the iterations in the while loop may continue if and only if both processes flip the same coin combination in an iteration. This occurs with probability 1/2 in every iteration. We now state the properties of this algorithm, whose proofs may be found in the original paper [87].

**Theorem 1.** *The two-process test-and-set implementation of [87] given in Figure 5.5 ensures the following properties.*

- *(Single Winner) In any execution, at most one process may return* 0*.*
- *(Winner-Loser Ordering) Given any* test-and-set *operation $\tau$ that returns* 1*, there exists another* test-and-set *operation $w$, starting before $\tau$ returns, such that $w$ either (i) returns* 0 *or (ii) does not complete.*
- *(Probabilistic Termination) Operation by correct processes terminate with probability* 1*.*
- *(Complexity) The algorithm has expected constant read-write step complexity. Given a constant $\alpha \geq 1$, the probability that a process performs more than $\alpha \log \ell$ reads and writes while running the algorithm is at most $1/\ell^2$.*

The first two processes imply that the implementation is *linearizable.*

**Proposition 2.** *The test-and-set algorithm of [87] is* linearizable.

**Randomized test-and-set for three processes.** In the next section, we will need a test-and-set object implementation that can be accessed by three processes. We will implement this from two two-process test-and-set objects $T_1$ and $T_2$. Assume that the object is accessed by three processes $p_1, p_2, p_3$, where each process knows its index $i \in \{1,2,3\}$. Processes $p_1$ and $p_2$ will participate in object $T_1$. The winner of $T_1$ will participate in $T_2$ with process $p_3$. The winner of object $T_3$ is the overall winner of the three-process test-and-set object. It is trivial to check that this implementation describes a correct three-process test-and-set. Note that the two-process test-and-set matches are decided in a wait-free manner, since a process wins automatically if the opponent does not show up.

## 5.2 The RatRace Algorithm

Now we are ready to present our adaptive one-shot test-and-set implementation. If $k$ is the contention in the current execution, any *test-and-set* operation on the object has step complexity $O(\log k)$ per process with high probability in $k$[1]. The algorithm pre-allocates $O(n^3)$ memory, and uses $O(k)$ memory with high probability. A sketch of the algorithm's structure can be found in Figure 5.8.

**Algorithm Structure.** Fix a constant $c \geq 3$. The algorithm is based on a binary tree structure, of height $c \log n + 1$, which we call the *primary tree*. Each node $v$ in this primary tree has two components: one randomized splitter object $RS_v$, and a three-process test-and-set object $T_v$. Both these objects are implemented as described in Section 5.1. Each randomized splitter $RS_v$ has two associated pointers, referring to splitter objects corresponding to the left and right children of node $v$. Thus, if node $v$ has children $\ell$ (left) and $r$ (right), the left pointer of $RS_v$ will refer to $RS_\ell$, while the right pointer refers to $RS_r$. Further, any process $p_i$ returning *left* from the randomized splitter $RS_v$ will call the *split* procedure of $RS_\ell$, while processes returning *right* will call the *split* procedure of $RS_r$.

Processes start at the root node of the primary tree, and proceed left or right (with probability $1/2$) through the tree until first returning *stop* at the splitter $RS_v$ associated to some node $v$. If a process reaches a leaf of the primary tree without having acquired a splitter, it accesses a *backup grid*, which we describe in the next paragraph. To simplify the exposition, assume that, in the execution we describe, all processes either obtained randomized splitters in the first tree, or crashed. Once it managed to obtain a splitter, the process tries to work its way up back to the root, through a series of three-process "tournaments," one at each splitter node. Each splitter in the primary tree has associated with it a three-player "tournament," which is played between the owner of the splitter (if any) and the winners of the three-player test-and-sets corresponding to the two child nodes of the splitter. A three-player test-and-set is decided as follows: the two child nodes first play each other (they are processes $p_1$ and $p_2$ in the three-process test-and-set described in Section 5.1). The owner of the current splitter plays the winner of the first two-process test-and-set (this process is $p_3$ in the three-process

[1]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 $O(\log n)$ factor in the running time of the algorithm.

```
Shared:
The primary tree PT, a binary tree of height $3\log n$. index is the index of the node in a
breadth-first-search labeling of the tree. Processes share a backup grid B of deterministic
splitters, accessed through the walk-backup-grid procedure. Its pseudocode is similar to that
of the primary tree, and is skipped. The winner from the primary tree meets the winner from
the backup grid in a final two-process test-and-set $T_D$. The register $R_D$ is initially false.
procedure test-and-set()
  if Resolved = true then return 1
  walk-down( PT.root )
procedure walk-down (v)
  res ← $RS_v$.split(id)
  if res = stop then
      name ← v.index
      val ← $T_v$.test-and-set()
      if val = 0 then
          walk-up( v.parent )
      else
          Resolved ← true
          return 1
  else
      if v is a leaf then
          walk-backup-grid(B)
      if res = left then
          walk-down(v.left)
      else walk-down(v.right)
procedure walk-up(v)
  res ← $T_v$.test-and-set()
  if res = 1 then
      Resolved ← true
      return 1
  if v.parent = null then
      decide()
  else walk-up( v.parent )
procedure decide()
  res ← $T_D$.test-and-set()
  if res = 1 then
      Resolved ← true
      return 1
  else
      return 0
```

Figure 5.6: Pseudocode of the one-shot $n$-process test-and-set object.

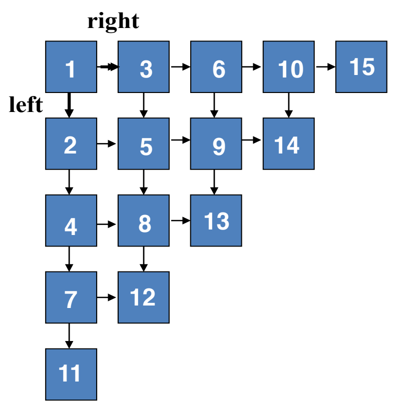

Figure 5.7: Structure of the backup grid.

test-and-set described in Section 5.1). Recall that each two-player match is decided using the randomized two-process test-and-set algorithm of Tromp and Vitànyi [87].

We say that a process *acquires* a splitter $s$ if it returns *stop* at the splitter $s$. So far, we have described a process's walk down the tree, and the corresponding walk up to the root in the case where every process acquires a splitter object in the primary tree. We now describe the procedure if a process "falls off" at a leaf in the primary tree.

**The Backup Grid.** The backup grid is an $n \times n$ grid of *deterministic* splitters, identical to that of Anderson and Moir [72], where the two children of a splitter are the splitter to its right, and the one below. Each process starts the backup algorithm at the top left splitter. As such, the structure guarantees that any correct process that accesses it eventually acquires a deterministic splitter. Just as in the previous case, once a process acquires a splitter, it tries to backtrack to the entry point through a series of three-player test-and-sets. The winner of the test-and-set corresponding to the entry splitter is also the winner of the backup grid.

**Decision.** The winner of the three-player test-and-set at the root of the primary tree plays the winner of three-process test-and-set the entry splitter in the backup grid. The winner of this last match returns 0. Every process that loses in a three-player test-and-set returns 1.

**Linearization.** To maintain the linearizability of the test-and-set object, a processes that loses any two or three player test-and-set writes *true* to a multi-writer-multi-reader *Resolved* register associated with the root of the primary tree, *before* returning 1. Processes read the register as the first step in their *test-and-set* invocation: if they read *true*, they immediately return 1.

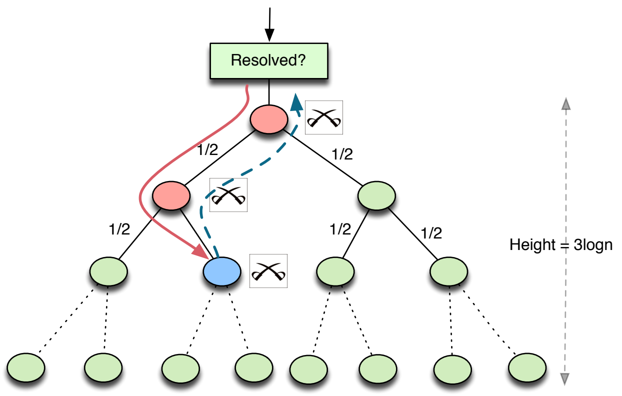

Figure 5.8: Structure of the *RatRace* protocol.

## 5.2.1 Analysis of the RatRace Algorithm

**Correctness.** We first check that the *RatRace* algorithm guarantees the correctness properties of the *test-and-set* object as stated in Section 5.2. The first step is the observation that the two-process and three-process test-and-set objects used by the algorithm are well-formed, i.e. they may only be accessed by the corresponding number of processes.

**Lemma 1.** *In any execution of the algorithm, a two-process test-and-set is accessed by at most two processes. A three-process test-and-set is accessed by at most three processes.*

*Proof.* This claim follows trivially by backward induction over the height of the primary tree, and over the depth of the backup grid.

For the primary tree, the key observation in the induction step is the following. Since at most one process may return *stop* at the splitter object corresponding to a node $v$, and at most two other distinct processes may win the test-and-sets corresponding to the children of the node (which are well-formed by the induction base), at most three distinct processes may access the three-process test-and-set corresponding to node $v$, and hence this object is well-formed.

The claim is identical for the backup grid. Therefore, at most two processes (the winner from the root of the primary tree and the winner from the root of the backup grid) may access the deciding two-process test-and-set $T_D$, so this object is well-formed as well. □

**Lemma 2.** *The* RatRace *implementation guarantees the following correctness properties.*

1. *(Single Winner) In any execution, at most one process may return* 0.
2. *(Winner-Loser Ordering) Given any* test-and-set *operation $\tau$ that returns* 1, *there exists another* test-and-set *operation $w$, starting before $\tau$ returns, such that $w$ either (i) returns* 0 *or (ii) does not complete.*

3. *(Probabilistic Termination) Every correct process returns with probability* 1.

*Proof.* For the *single winner* property, first notice that Lemma 1 ensures that all the test-and-sets used by the algorithm are well-formed, therefore these objects ensure the properties of a correct implementation.

In particular, this is true for the “deciding" two-process test-and-set $T_D$, therefore a single process may return 0 from the $T_D$, since this object ensures the *single winner* property. Therefore, *RatRace* also ensures *single winner.*

For the *winner-loser ordering* property, we consider an arbitrary execution $\mathscr{E}$ of the protocol, and a process $p_\ell$ executing operation $\tau$ that returns 1 in $\mathscr{E}$. We prove that there has to exist an operation $w$ by process $p_w$ which starts before $\tau$ and may not return 1.

By the structure of the protocol, there are two cases: either the process $p_\ell$ returned 1 because it lost in a two- or three-process test-and-set $T_1$, or because it read *Resolved* = *true*. We start by considering the first case.

If $p_\ell$ returned 1 because it lost a low-level test-and-set operation on object $T_1$, then, since $T_1$ ensures the winner-loser ordering property, there has to exist an operation $w_1$ by some process $p_1$ which started before $\tau$ returned from the invocation on $T_1$, and may not return value 1 from $T_1$. If $w_1$ does not return from its invocation on $T_1$, then we are done, since $w_1$ does not return from its invocation on *RatRace* either. Otherwise, there are two cases. If $p_1$ returns 0 from its invocation on *RatRace*, then we are done.

Otherwise, $p_1$ returns 1, and therefore it must have lost a two- or three-process test-and-set $T_2$ as part of the algorithm. We apply the same rationale inductively, and we obtain that there exists a *maximal* chain of processes $p_1, p_2, \ldots, p_j$, with operations $w_1, w_2, \ldots, w_j$ on *RatRace*, such that, for any $1 \leq i \leq k-1$, process $p_i$ “lost" to process $p_{i+1}$ in a two- or three-process test-and-set $T_i$. Notice that the length of this chain is at most $k-1$, where $k$ is the number of processes participating in the current execution.

We assume that this process chain $p_\ell, p_1, \ldots, p_j$ is *maximal*, in that process $p_j$ does not lose to another process $p_{j+1}$ on an object $T_{j+1}$ in this execution. Therefore, there are only two possibilities for $p_j$: either $p_j$ is correct and returns 0 from *RatRace*, or $p_j$ crashes.

Therefore, we only need to prove that $p_j$'s *test-and-set* operation $w_j$ started *before* $p_\ell$'s *test-and-set* operation returns. This is ensured by the *Resolved* bit mechanism. Recall that, upon returning 1, a process first sets the *Resolved* bit to *true*. Therefore, by the atomicity of the *Resolved* register, any operation that starts *after* $p_\ell$'s operation ends will read *Resolved* = *true*, and therefore automatically return 1. Therefore, operation $w_j$ cannot start after $p_\ell$'s operation ends. Hence, we have found an operation $w_j$, starting before $\tau$ returns, such that it either returns 0 or does does not complete, proving our claim in this case.

The second case is when $p_\ell$ returned because it read *true* from the *Resolved* register. Since the initial value of this register is *false*, there exists another operation $p_1$ invoking operation $w_1$ which wrote *true* to *Resolved*. By the structure of the algorithm, this operation either returns 1 since it lost a test-and-set object, or does not terminate. In the latter case, the claim follows. In the former case, by the previous argument, there exists another operation $w$, which starts before operation $w_1$ writes *true* to *Resolved*, and either returns 0 or does not complete. Since this operation starts before $p_1$ writes *true* to *Resolved*, it must also start before $p_\ell$ returns from $\tau$, which concludes the proof of this claim.

Finally, for the *probabilistic termination* property, notice that the algorithm uses only wait-free elements and the two-process test-and-set algorithm of [87], which ensures termination with probability 1. Therefore, every *test-and-set* invocation in *RatRace* will also terminate with probability 1.

□

The immediate consequence of this Lemma is that the *RatRace* algorithm implements a linearizable test-and-set object, ensuring termination with probability 1.

**Lemma 3** (Linearization)**.** *The* RatRace *algorithm is* linearizable*: for every execution of* RatRace*, there exists a total order over all the complete* test-and-set *operations together with a subset of the incomplete* test-and-set *operations such that every operation is immediately (atomically) followed by a response, and the sequence of operations given by that total order is consistent with a sequential execution of a test-and-set object.*

*Proof.* Consider an arbitrary execution $\mathscr{E}$ of an instance $T$ of *RatRace*. We show that $\mathscr{E}$ is linearizable.

We may construct the linearization order for $\mathscr{E}$ as follows. Consider the set of operations that result in a 1 response indication in $\mathscr{E}$. We linearize these operations based on the order in which the respective processes wrote in the *Resolved* register. We consider $op$ to be the *first* losing operation in this linearization order, and let $p$ be the respective caller. Notice that, once $p$ returns 1 from $T$, every process accessing the object after $p$ returns will receive a 1 indication (or crashes).

On the other hand, notice that, by Lemma 2, for any response of 1 returned by an operation $\tau$ on $T$, there exists an operation $w$ such that either (1) $w$ is a completed operation that returns 0, or (2) $w$ is an incomplete operation (a crash), that starts *before* the call that generated the 1 response. Also, there may exist at most one such operation that returns 0. In particular, let $w$ be this operation corresponding to operation $op$ defined above. Therefore, $w$ may be linearized *before* operation $op$, and either returns 0 or crashes. We have thus defined a total order on all completed operations, which matches the sequential specification of the test-and-set object. □

**Performance.** We now analyze the performance of *RatRace*. Recall that $k$ denotes the number of processes that enter the *RatRace* in an execution $\mathscr{E}$, i.e. the total contention. The next result states that, with high probability, every process acquires a splitter in the primary tree. As a consequence of this fact, for the rest of the performance analysis, we will assume that all processes acquire nodes in the primary tree, since the backup case is extremely unlikely. The proof is similar to the analysis in [25], Lemma 8.

**Lemma 4.** *For any $c \geq 3$ constant, the probability that there exists a process $q$ that does not acquire a randomized splitter in the primary tree of the* RatRace *object is at most $1/n^{c-2}$.*

*Proof.* Let $q$ be a process that does not manage to acquire any splitter in the primary tree. Hence, $q$ did not manage to acquire the leaf splitter it reached in the primary tree. Since, by the properties of a splitter, a process always acquires a splitter if it accesses it alone, this implies that another process $q'$ accessed the same leaf splitter. However, the leaf splitter is accessed by $q$ as a consequence of $c \log n - 1$ random choices of bits, i.e. the choices of left/right direction at every internal node. Hence the process $q'$ must have performed the exact same random choices at internal nodes as process $q$. Fix a process $q'$ such that this event occurs. Since the choices of each process are independent, the probability that the event occurs is upper bounded by

$$\left(\frac{1}{2}\right)^{(c \log n+1)-1} = 1/n^c.$$

We take the union bound over all possible $k \leq n$ processes, and obtain that the probability that there exists a process that performs exactly the same random choices as $q$ is at most $k/n^c \leq 1/n^{c-1}$. This gives us the probability that, for a given process $q$ that accesses a leaf splitter, there exists another process that accesses the same leaf.

Taking the union bound over all $n$ possible choices for $q$, it follows that the probability that there exists *some* process $p$ that "falls off" the primary tree is at most $1/n^{c-2}$, as desired. □

We now bound the step complexity of the protocol. First, we say that a node $v$ in the primary tree is *marked* if the splitter at $v$ is accessed during the execution by some process. Let the *active primary tree AT* denote the minimal subtree of the primary tree containing all splitters that are marked in the execution. We give a probabilistic bound on the height of the primary tree, which will also bound the number of steps a process may take in the execution.

**Lemma 5.** *For any $c \geq 1$, the height of the primary tree is at most $(c+2)\log k+1$, with probability at least $1 - 1/k^c$.*

*Proof.* Consider the walk $w$ that a process $p$ performs until acquiring a splitter. At each step, this walk $w$ can either stop, if the process acquires the splitter, or continue, if there exists another process $q$ that performs the same walk up to the current node, and accesses the splitter object at the end of $w$ concurrently with $p$. The probability that this event occurs depends on the length of the walk $w$, as follows.

Assume that the walk $w$ by process $p$ has length $\geq (c+2)\log k + 1$, with $c \geq 1$. Then, by the properties of a splitter, there has to exist (at least) one other process $q$ that accesses the splitter at position $(c+2)\log k$ in the walk, concurrently with $p$. Necessarily, the process $q$ must have made the same random choices as $p$, and is scheduled by the adversary so as to block $p$ from acquiring the last splitter in $w$. The probability that a given process $q$ makes the same $(c+2)\log k$ random choices as $p$ leading to the last node in $w$ is at most

$$\left(\frac{1}{2}\right)^{(c+2)\log k} = \left(\frac{1}{k}\right)^{c+2}.$$

Therefore, by a union bound, the probability that *there exists* a process $q$, among the $k-1$ other processes, making the same random choices as $p$, is at most $\left(\frac{1}{k}\right)^{c+1}$. Finally, by another union bound over all $k$ paths, one for each process, we obtain that the probability that there exists a path of length more than $(c+2)\log k + 1$ is at most $\left(\frac{1}{k}\right)^{c}$, as desired. □

Next, we look at the read-write complexity of the two-process test-and-set algorithm of Tromp and Vitànyi [87] that we use to decide the two-process games. The following bounds follow from an analysis of the algorithm. Its proof is given in [87], Theorem 5.13.

**Lemma 6** (Tromp and Vitànyi [87])**.** *The randomized two-process test-and-set algorithm of [87] has expected read-write complexity of* $11$ *steps. For any integer* $\ell \geq 1$*, the probability that a process takes more than* $11 \cdot \ell$ *steps as part of the protocol is at most* $(1/2)^{\ell-1}$.

The next result analyzes the step complexity of *RatRace*.

**Lemma 7** (Step Complexity)**.** *The* RatRace *algorithm uses* $O(\log k)$ *steps per process, with high probability in* $k$*. Hence, the total step complexity is* $O(k \log k)$*, with high probability in* $k$.

*Proof.* Without loss of generality, we analyze the number of steps performed by a winning process. First note that, by Lemma 4, it is enough to bound the complexity in the case where the process only accesses the primary tree, since the other case occurs with negligible probability.

By Lemma 5, since each randomized splitter takes at most 8 steps, the process performs $8((c+2)\log k + 1)$ steps when going down the tree in order to acquire a randomized splitter, with probability at least $1 - (1/k)^c$.

Also, when climbing back up, the process plays up to $m = 2(c+2)\log k + 1$ two-player test-and set games, with high probability (where the extra factor of two comes from the fact that each node contains a three-player test-and-set, composed of two-player test-and-sets). We now bound the number of steps that a process may take as part of these $m$ matches.

For $i = 1, \ldots, m$, let $Y_i$ be the random variable counting the number of 11-step rounds that the process executes in the two-process test-and-set with index $i$ on its walk back to the root. Lemma 6 ensures that each such random variable is geometrically distributed, with probability

$\Pr(Y_i = \ell) = (1/2)^\ell$. The total number of steps that the process performs is upper bounded by a constant times $Z = \sum_{i=1}^m Y_i$.

The random variable $Y$ can be seen as a constant times the number of Bernoulli trials with probability $1/2$ until there are $m$ successes, i.e. the distribution of $Y$ is a negative binomial distribution. Thus,

$$\Pr(Y = y) = \binom{y-1}{m-1}\left(\frac{1}{2}\right)^y.$$

For $y \geq 5m$, we have that

$$\frac{\Pr(Y = y+1)}{\Pr(Y = y)} = \frac{y}{2(y-m+1)} \leq \frac{5}{8}.$$

Therefore, for $y \geq 5m$, $Pr(Y \geq 5m)$ can be upper bounded by a geometric series, and we get

$$\Pr(Y \geq 5m) \leq \frac{8}{3}\Pr(Y = 5m) \leq \frac{8}{3}\binom{5m-1}{m-1}\left(\frac{1}{2}\right)^{5m} \leq \frac{8}{3}\left(\frac{(5m-1)e}{m-1}\right)^{m-1}\left(\frac{1}{2}\right)^{5m} \leq \left(\frac{1}{2}\right)^m.$$

Therefore, since $m = 2(c+2)\log k + 1$, we obtain that the probability that a specific process $p$ takes more than $11(2(c+2)\log k + 1)$ steps in the execution is at most $(1/k)^{22(c+2)}$. By taking a union bound over the $k$ processes, we obtain that the probability of any process taking more than $O(\log k)$ steps in the ascent to the root is at most $(1/k)^\beta$, for some constant $\beta \geq 2$, as desired.

Summing up over the descent to acquire a splitter and the ascent towards the root, we obtain that every process performs $O(\log k)$ steps, with probability at least $1 - (1/k)^\alpha$, for a constant $\alpha = \min(c, \beta) - 1$. The total step complexity upper bound is straightforward. □

## 5.3 RatRace as a renaming algorithm

Consider a breadth-first search labeling of the primary tree, and a labeling of the backup deterministic splitter grid, as given in Figure 5.7. We transform *RatRace* into a renaming algorithm as follows.

- Each process that acquires a randomized splitter in the primary tree returns the label of the corresponding node in the breadth-first labeling, on line 9.
- Each process that acquires a deterministic splitter returns the label of the corresponding node plus an additive factor of $2^{c\log n+1} = 2n^c$. (This factor will ensure that the primary and backup namespaces do not clash.)

Lemma 5 ensures that the namespace returned by this algorithm is of size $2k^c$, with probability at least $1 - 1/k^c$. Lemma 7 ensures that the algorithm has step complexity $O(\log k)$, with probability at least $1 - 1/k^c$. We formalize this as follows.

**Lemma 8** (*RatRace* Renaming)**.** *The* RatRace *algorithm yields an adaptive renaming algorithm ensuring a namespace of size $O(k^c)$ in $O(\log k)$ steps, both with high probability in $k$.*

*Proof. Termination* with probability 1 follows from Lemma 2. The upper bounds on step complexity follow from Lemma 7. *Name uniqueness* follows from the fact that no splitter may be acquired by two processes. Finally, notice that Lemma 7 ensures that a process stops at a splitter of height $b \log k$, with probability at least $1 - (1/k)^\alpha$, for constants $\alpha$ and $b \geq 1$. Since the names assigned are from a breadth-first search labeling of the tree, the names assigned are from 1 to $k^b$, with probability $1 - (1/k)^\alpha$. This completes the proof. □

## 5.4 Related Work

The test-and-set instruction has been present in hardware for several decades, as a simple means of implementing mutual exclusion. Herlihy [56] showed that this object has consensus number 2, i.e. that it can implement consensus for two processes, but it cannot implement consensus for three processes.

Several references studied wait-free randomized implementations of test-and-set. References [82, 55] presented implementations with super-linear step complexity. (Randomized consensus algorithms also implement test-and-set, however their step complexity is at least linear [18].) The first randomized implementation with logarithmic step complexity was by Afek et al. [3], who extended the mutual exclusion tournament tree idea of Peterson and Fischer [81], where the tree nodes are two-process test-and-set (consensus) implementations as presented by Tromp and Vitanyi [87]. Their construction has expected step complexity $O(\log n)$. This technique is made adaptive by the *RatRace* protocol, originally published in [9]. The *RatRace* protocol has been used as a sub-procedure for obtaining sub-logarithmic test-and-set against a weak adversary [6].

References [53, 52] give deterministic test-and-set and compare-and-swap implementations with constant complexity in terms of *remote memory references* (RMRs), in an asynchronous shared-memory model with no process failures (by contrast, our implementation is wait-free; its step complexity is also an upper bound on its RMR complexity, although we do not explicitly optimize for the RMR complexity metric). Reference [3] gives a procedure to transform a single-use test-and-set protocol into a multi-use, resettable version. This procedure was generalized in [6], and also applies to the *RatRace* protocol.

# 6 Randomized Strong Renaming in Logarithmic Time

In the previous chapter, we have presented a randomized implementation for a *test-and-set* object, which elects one winner from a set of $k$ participating processes. In this chapter, we present algorithms for *strong* renaming, which ensures a namespace that matches the number of participants. If test-and-set can be seen as a single-winner distributed tournament object, renaming can be seen as a generalized distributed tournament, which assigns a unique rank to *each* participant.

We give two algorithms. The first is *adaptive*, i.e. its namespace size always matches the contention $k$ in the execution. The second is non-adaptive, i.e. its namespace size is $n$, but uses linear space, assuming test-and-set operations are present in hardware.

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

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 will obtain an algorithm with worst-case step complexity $\Theta(\log M)$, which does not adapt to the number of participants $k$ (in fact, this dependency is not allowed by the formal specification of the renaming problem [17]). The second obstacle is that a regular sorting network construction has a fixed number of input ports. Since we would like to avoid assuming any bound on the contention $k$ in the execution, we need to build a sorting network that "extends" its size as the number of 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 $k$[1]. This is the first algorithm that achieved sub-linear complexity. We then show how to use it to obtain randomized implementations of counter and fetch-and-increment objects. Further, the existence of this algorithm will have important implications from the point of view of *lower bounds* for the renaming problem, as we will see in Chapter 7.

**Bit batching.** The second algorithm assigns unique names to processes by repeatedly sampling over batches of test-and-set bits of decreasing size. More precisely, we split a sequence of $n$ registers in batches of exponentially decreasing size, such that the first batch contains the first half of the registers, the second contains the next quarter, and so on, until we reach batches of size $\Theta(\log n)$. Processes perform test-and-set operations on $\Theta(\log n)$ registers in each batch, until first winning a test-and-set. A careful analysis shows that, somewhat surprisingly, every process obtains a test-and-set object before completing its test-and-set operations on the last batch, with high probability.

In Section 6.1, we give some background on sorting networks and counters. In Section 6.2, we show how to solve renaming using a sorting network of fixed size. We give an extensible sorting network construction in Section 6.3.1, and show how to assign input ports to this network in an adaptive manner in Section 6.3.2. We present some applications to counting objects in Section 6.4. The *BitBatching* algorithm is presented in Section 6.5. Finally, we give an overview of related work in Section 6.6.

## 6.1 Preliminaries

### 6.1.1 Sorting Networks

A sorting network is a *comparison network* that always sorts its inputs, therefore we will start with the description of comparison networks. We follow the description from [41]. A *comparison network* is a network composed solely of wires and comparators. A *comparator* is a device with two inputs, $x$ and $y$, and two outputs, $x'$ and $y'$. A comparator has the property that $x' = \min(x, y)$, and $y' = \max(x, y)$. See Figure 6.1 for an illustration. We can thus think of a comparator as sorting its two inputs (in decreasing order).

A *wire* transmits a value through the network. Wires can connect the output of one comparator to the input of another, otherwise they are either network input wires or network output wires. For $n \geq 1$, the input wires $a_1, \ldots, a_n$ take values as inputs, while the output wires $b_1, \ldots, b_n$ produce the results computed by the comparison network. We describe an input sequence $a_1, \ldots, a_n$, and an output sequence $b_1, \ldots, b_n$, referring to the values of the input and output wires. (Thus, we use the same name for both the wire and the value it carries.)

[1]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 $O(\log n)$ factor in the running time of the algorithm.

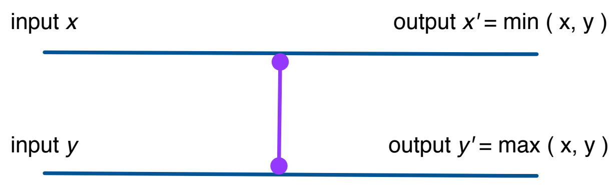

Figure 6.1: Structure of a comparator.

Figure 6.2 describes a comparison network with 4 inputs and 4 outputs. The *horizontal* lines are wires, while comparators are stretched vertically. (Note that a horizontal line does not represent a single wire, but rather a sequence of distinct wires connecting comparators.) The main requirement for connecting comparators is that the graph of interconnections must be *acyclic*: if we trace a path from the output of a given comparator to the input of another, to another input etc., the path we trace must never cycle back on itself or go through the same comparator twice. Thus, a comparison network can be drawn with network inputs on the left and network outputs on the right: data move through the network from left to right.

Under the assumption that each comparator takes unit time, and transmission through a wire takes zero time, the “running time" of a comparison network is the time it takes for all output wires to receive their values. Informally, this is the largest number of comparators that any input can pass through as it travels from an input wire to an output wire. More formally, we define the *depth* of a wire as follows. An input wire has depth 0. A comparator that has two input wires with depths $d_x$ and $d_y$ will have depth $\max(d_x, d_y) + 1$. Because there can be no cycles of comparators, the depth of a wire is well defined. The depth of a comparator is the depth of its output wires. Finally, the *depth* of a comparison network is the maximum depth of an output wire. For example, the comparison network of Figure 6.2 has depth 3.

A *sorting network* is a comparison network for which the output sequence is monotonically increasing, i.e. $b_1 \leq b_2 \leq \ldots \leq b_n$, for *every* input sequence. The comparison network in Figure 6.2 is a sorting network.

A useful fact when proving that a comparison network sorts is the *zero-one principle* [41].

**Theorem 2** (Zero-One Principle)**.** *If a comparison network with n inputs sorts all $2^n$ possible sequences of 0's and 1's correctly, then it sorts all sequences of arbitrary numbers correctly.*

The key challenge when building a correct sorting network is using the minimal number of comparators that still ensures correctness. A landmark result by Ajtai et al. [5] showed that there exist sorting networks of logarithmic depth.

**Theorem 3** (AKS Sorting Networks)**.** *For any $n \geq 2$, there exists a sorting network with $O(\log n)$ depth.*

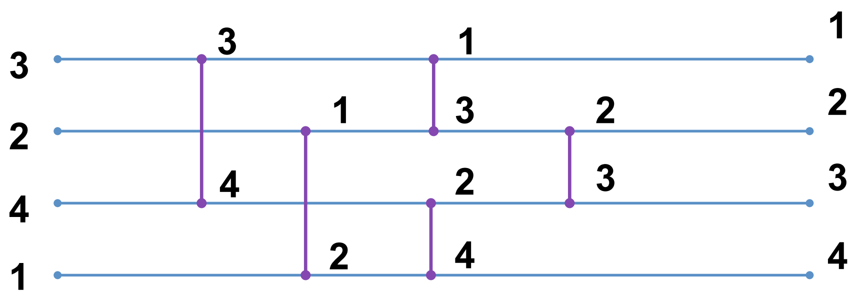

Figure 6.2: Structure and execution of a sorting network.

## 6.1.2 Counter and Max-Register Objects

The shared *counter* object maintains a value $V$, initially 0, and exports operations *increment* and *read*. The *increment* operation atomically increments $V$ by 1, and returns *success*, while the *read* operation operation returns the current value of the object.

The *fetch-and-increment* object also maintains a value $V$, initially 0, and exports a single *fetch-and-inc* operation, which atomically increments the returns the current value of the object by 1, and returns the previous (un-incremented) value of the object. Randomized versions of these objects maintain correctness in all executions, but ensure termination with probability 1.

The *max-register* is a shared object maintaining a value $V$, initially 0, which records the highest value ever written to it. The *max-register* defines a default maximal value $v_{\max}$ which it may store. Aspnes et al. [12] gave an implementation of a max-register with logarithmic complexity in $v_{\max}$.

**Theorem 4** (Aspnes et al. [12]). *There exists a linearizable, deterministic, wait-free max-register construction, where each operation has cost* $O(\log v_{\max})$.

In the same paper, Aspnes et al. introduce the notion of monotone consistency, which is a weakening of linearizability. For example, a counter data structure (as defined above) is *monotone consistent* if the following hold.

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

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

Figure 6.3: Pseudocode for executing a renaming network.

## 6.2 Renaming using a Sorting Network

In this section, we present a solution for adaptive strong renaming using 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 6.3. Note that the bound on $M$ implies a bound on $n$, the maximum number of participating processes.

### 6.2.1 Renaming Networks

We start from an arbitrary sorting network with $M$ input and output ports, in which we replace the comparator modules with two-process test-and-set objects, implemented using the algorithm of Tromp and Vitányi [87]. 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 6.3). Also part of the process's algorithm is the blueprint of a renaming network with $M$ input ports, which is the same for all participants.

We use the renaming network to solve adaptive tight renaming as follows. (Please see Figure 6.3 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 moves "down," i.e. follows output port $y'$. Each process continues until it reaches an output port $b_\ell$. The process returns the index $\ell$ of the output port $b_\ell$ as its output value. See Figure 6.4 for a simple illustration of a renaming network execution.

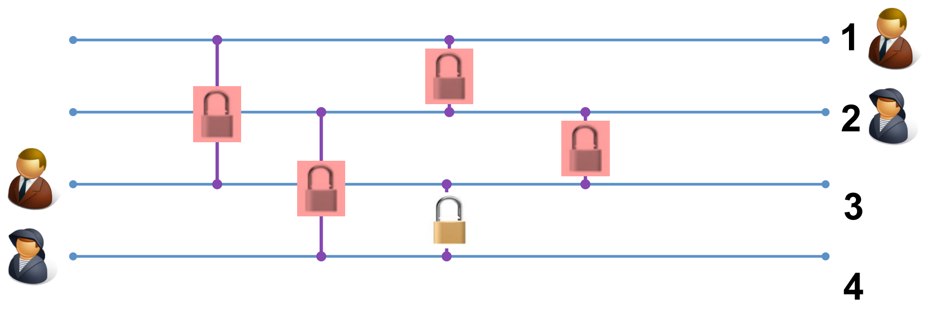

Figure 6.4: Execution of a renaming network.

### 6.2.2 Renaming Network Analysis

In the following, we show that the renaming network construction solves adaptive tight 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 5** (Renaming Network Construction)**.** *The renaming network construction solves tight adaptive renaming, ensuring termination with probability* 1.

*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.

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

*Proof.* Recall that each renaming network is obtained from a sorting network. Therefore, for any renaming network, we can maintain the definitions of network and wire depth as for a sorting network, given in Section 6.1.1. 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 $d \geq 1$. 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 $d$ 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 with probability 1 follows from the fact that the base sorting network has finite depth and, by definition, contains no cycles, and from the *probabilistic termination* property of the two-process test-and-set implementation [87]. We prove name uniqueness and namespace tightness by ensuring the following claim.

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

The proof is based on a simulation argument from an execution of a renaming network to an execution of a sorting network. We start from an arbitrary execution $\mathscr{E}$ 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 $\mathscr{E}$. Each process $p_i \in P$ has 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 $\mathscr{E}$, 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 $\mathscr{E}$, 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 $\mathscr{E}$, 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 $\mathscr{E}$, 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 1, 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 $\mathscr{E}$ 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 $\mathscr{E}$. Therefore, the process with associated value 0 always wins the test-and-set.
3. If $p$ and $q$ are both in $Q$, 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 1, no two input values may be sorted to the same output port, we first obtain that the output port indices that the processes in $P$ return are unique. For namespace tightness, recall that we have obtained an execution of a sorting network with $M$ input values, $M-k$ of which, 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 $\mathscr{E}$ 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 $\mathscr{E}$. The final step is to notice that, in $\mathscr{E}$, 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 $\mathscr{E}$. 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 depth, whose existence is ensured by Theorem 3.

**Corollary 1** (Complexity)**.** *The renaming network obtained from an AKS sorting network [5] with $M$ input ports solves the strong adaptive renaming problem with $M$ initial names, guaranteeing termination with probability* 1 *and name uniqueness in all executions, using* $O(\log M)$ *test-and-set operations per process in the worst case.*

*Proof.* The fact that this instance of the algorithm solves strong adaptive renaming follows from Theorem 5. 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 width is $O(\log M)$. □

## 6.3 A Strong Adaptive Renaming Algorithm

We present an algorithm for adaptive tight 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

### 6.3.1 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 $k$. 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 $B'_1, B'_2, \ldots B'_m, A'_1, A'_2, \ldots A'_m$. Internally, insert $B$ between $A$ and $C$ by connecting outputs $A'_1, \ldots, A'_{k-\ell}$ to inputs $B_{\ell+1}, \ldots, B_k$; and outputs $B'_{\ell+1}, \ldots B'_k$ to

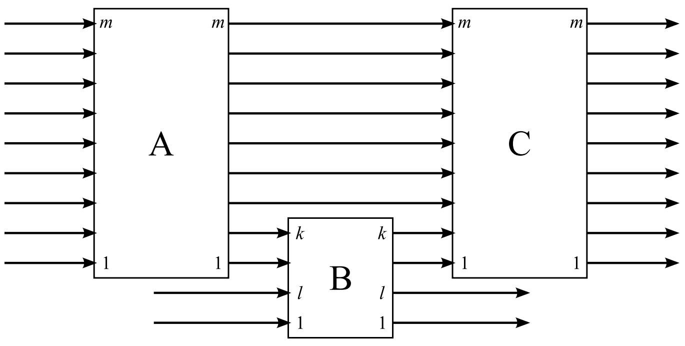

Figure 6.5: One stage in the construction of the adaptive sorting network.

inputs $C'_1, \dots C'_{k-\ell}$. The remaining outputs of $A$ are wired directly across to the corresponding inputs of $C$: outputs $A'_{k-\ell+1}, \dots, A'_m$ are wired to inputs $C_{k-\ell+1}, \dots, C_m$. (See Figure 6.5.)

**Lemma 9.** *The network $ABC$ constructed as described above is a sorting network.*

*Proof.* The proof uses the well-known Zero-One Principle, given in Theorem 2: 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 $B_1 \dots B_\ell$ and $A_1 \dots A_m$. 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 $B'_1 \dots B'_{z_B+\max(k-\ell,z_A)}$. An additional $z_A - \max(k-\ell, z_A)$ zeros propagate directly from $A$ to $C$.

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

- Case 1: $z_A \le k-\ell$. Then $B$ gets $z_B + z_A$ zeros (all of them), sorts them to its lowest outputs, and those that reach outputs $B'_{\ell+1}$ and above 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 $\ell \le k/2$, $z_B + k - \ell \ge k/2 \ge \ell$, and B sends $\ell$ zeros out its direct outputs $B'_1 \dots B_\ell$. 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 10.** *If a value $v$ is supplied to one of the inputs $B_1$ through $B_\ell$ in the network $ABC$, and is one of the $\ell$ smallest values supplied on all inputs, then $v$ never leaves $B$.*

*Proof.* Immediate from the construction and Lemma 9; $v$ 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 polylog $N$ depth when truncated to the first $N$ positions. We assume that we are using a construction of a sorting network that requires at most $a \log^c n$ depth to sort $n$ values, where $a$ and $c$ are constants. For the AKS sorting network [5], we have $c = 1$; for constructible networks (e.g., the bitonic sorting network [64]), we have $c = 2$.

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

Given $S_k$, construct $S_{k+1}$ by appending two sorting networks $A_{k+1}$ and $C_{k+1}$ with width $w_k^2 - w_k/2$, and attach them to the top half of $S_k$ as in Lemma 9, setting $\ell = w_k/2$.

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

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

We can in fact state something stronger:

**Theorem 6.** *Each of the networks $S_k$ constructed above is a sorting network, with the property that any value that enters on the $n$-th input and leaves on the $m$-th output traverses $O(\log^c \max(n,m))$ comparators.*

*Proof.* That $S_k$ is a sorting network follows from induction on $k$ using Lemma 9.

For the second property, let $S_{k'}$ be the smallest stage in the construction of $S_k$ to which input $n$ and output $m$ are directly connected. Then $w_{k'-1}/2 < \max(n,m) \le w_{k'}/2$, which we can rewrite as $2^{2^{k'-1}} < 2\max(n,m) \le 2^{2^{k'}}$ or $k' - 1 < \lg\lg \max(n,m) \le k'$, implying $k' = \lceil \lg\lg \max(n,m) \rceil$. By Lemma 10, the given value stays in $S_{k'}$, meaning it traverses at most $d_{k'} = O\left(2^{ck'}\right) = O\left(2^{c\lceil \lg\lg \max(n,m)\rceil}\right) = O\left(\lg^c \max(n,m)\right)$ comparators. □

### 6.3.2 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 6.3.1. (Recall that a renaming network is a sorting network in which all comparators are replaced with two-process test-and-set objects.)

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.

### 6.3.3 Technical Notes

**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 [5] of $O(\log M)$ depth 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 [64]. On the other hand, since the construction accepts any sorting network as basis, we can use Batcher's bitonic sorting network [64], 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.

### 6.3.4 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 7** (Namespace Boosting). *Given any renaming algorithm $A$ ensuring namespace $M$ with worst-case step complexity $C(A)$, the renaming network construction yields an algorithm $S(A)$ ensuring* strong *renaming with worst-case expected step complexity $C(A) + O(\log M)$. The number of steps that a process performs in the renaming network is $O(\log M)$ with high* *probability in $k$. Moreover, if $A$ is* adaptive*, then the algorithm $S(A)$ is also adaptive.*

*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 an input port of the adaptive renaming network between 1 and $M$. We truncate the renaming network after the first $M$ input ports. By Theorem 6, we obtain that the original comparison network truncated after the first $M$ input ports is in fact a sorting network. Since $A$ is a correct renaming algorithm, no two processes may enter the same input port. Therefore, using Theorem 5, we obtain that each process returns a unique output port index between 1 and $k$. This 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 notice that the entire construction is adaptive, since the adaptive renaming network assumes no bound on $n$, and its complexity and output namespace size depend only 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 1, we obtain that worst-case expected step complexity in the second stage of the protocol is $O(\log M)$. Therefore, the worst-case expected step complexity of the transformation 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 notice that, since the names returned by the algorithm $A$ are unique, we have that $M \geq k$. In the worst case, 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 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 $k$. □

As an example, we substitute the generic algorithm $A$ with the *RatRace* renaming algorithm presented in Section 5.3, to obtain a strong renaming algorithm with logarithmic step complexity. First, recall the properties of the *RatRace* renaming algorithm.

**Proposition 3** (*RatRace* Renaming)**.** *The* RatRace *algorithm yields an adaptive renaming algorithm ensuring a namespace of size $O(k^c)$ in $O(\log k)$ steps, with high probability in $k$.*

We now prove the following.

**Corollary 2.** *There exists an algorithm $T$ such that, for any $k \geq 1$, $T$ 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 7 with *RatRace* renaming. We obtain a correct adaptive strong renaming algorithm.

For the upper bounds on complexity, by Lemma 3, 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 $c \geq 2$. 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 7, with probability at least $1 - 1/k^c$, this is at most

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

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

## 6.4 Applications to Counting

### 6.4.1 A Monotone-Consistent Bounded Counter

We now build a monotone-consistent counter algorithm based on the strong adaptive renaming algorithm in Section 6.3. The algorithm exports *read* and *increment* operations, and has a bounded maximum value $v_{\max}$.

**Description.** The processes share an adaptive renaming object implemented using the construction in the previous section, and a linearizable max-register, implemented using the construction of Aspnes et al. [12], whose properties are given in Section 6.1.

For the *increment* operation, a process acquires a new name from the adaptive renaming object. It then writes the newly obtained name to the max-register and returns. For the *read* operation, the process simply reads the value of the max-register and returns it.

**Analysis.** The counter object has the following properties.

**Lemma 11** (Counter Properties)**.** *The counter implementation is monotone consistent, and has expected step complexity $O(\log v)$ per increment, where $v$ is the number of* increment *operations started before the operation returns. A* read *operation has cost $O(\min(\log v, O(n)))$.*

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

(1) There exists a total ordering $<$ on the *read* operations such that if an operation $R1$ finishes before some operation $R2$ starts, then $R1 < R2$, and if $R1 < R2$, then the value returned by $R1$ is less than or equal to the value returned by $R2$. For this, we order the read operations by their linearization points when reading the max-register object. This ordering clearly has the required properties.

(2) The value $v$ returned by a read is always $\geq$ the number of completed *increment* operations. Let $y$ be the number of completed *increment* operations. Notice that each completed operation obtains a unique name, and writes it to the max-register (this holds also if a single process performs multiple *increment* operations). It then follows that the value in the max-register at the time of the read is at least $y$.

(3) The value $v$ returned by a read is always $\leq$ the number of started *increment* operations. Let $z$ be the number of started *increment* operations. Assume for contradiction that a process returns a value $v$ which is larger than $z$. In this case, there must exist a process that returned a name which is strictly larger than the number of name requests on the adaptive renaming object. This contradicts the *adaptive* property of the object.

Therefore the counter object is monotone consistent. For the complexity bound on the *increment* operation, notice that the complexity of the first stage of the adaptive renaming protocol is $O(\log v)$, and the number of temporary names is $O(\text{ poly } v)$ with high probability. It then follows that the complexity of the adaptive renaming object is $O(\log v)$ in expectation, and $O(\log^2 v)$ with high probability in $v$. By the properties of the max-register, it follows that that the complexity of an *increment* operation is $O(\log v)$. The complexity of the *read* operation is the same as the complexity of the max register. □

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

### 6.4.2 Linearizable Bounded-Value Fetch-and-Increment

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

**Description.** The outline of the construction is as follows. We first use the tight adaptive renaming protocol to build a linearizable $\ell$-test-and-set object, which generalizes a standard test-and-set object by providing $\ell$ winners instead of a single one. We implement such an object by having processes run the adaptive tight renaming algorithm and return *true* if and

```
Shared: boolean doorway, initially open
procedure $\ell$-test-and-set()
if O.doorway = closed then
    return false
else
    name ← tight-renaming()
    if name ≤ $\ell$ then return true
    else
        O.doorway ← closed
        return false
```

Figure 6.6: The $\ell$-test-and-set implementation.

```
Shared: test, an $\ell/2$-test-and-set object
left, an $\ell/2$-valued f&inc object
right, an $\ell/2$-valued f&inc object
procedure $\ell$-fetch-and-increment()
  if $\ell = 0$ then return 0
  if $\ell/2$-test-and-set(O.test) then
      return fetch-and-increment(O.left)
  else
      return $\ell/2$ +
      fetch-and-increment(O.right)
```

Figure 6.7: The $\ell$-fetch-and-increment object.

only if their acquired name is at most $\ell$. To ensure this is linearizable, we protect the renaming protocol with a doorway bit, which guarantees that processes arriving after some process returns *false* cannot prevent a process that started the operation earlier from winning.

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

**Analysis.** We begin by formally defining an $\ell$-test-and-set object.

**Definition 6.** *An* $\ell$-test-and-set *object O supports one type of operation which returns either* true *or* false. *The sequential specification of the object is that the first* $\ell$ *invocations of the operation return* true *and the rest return* false.

Our implementation of an $\ell$-test-and-set object is given in Figure 6.6. The following lemma shows correctness of our implementation. Intuitively, any operation that starts late sees the doorway closed, therefore must return *false*.

**Lemma 12.** *Procedure $\ell$-test-and-set in Figure 6.6 implements a linearizable $\ell$-test-and-set.*

*Proof.* By correctness of the adaptive tight renaming algorithm, $\ell$ processes obtain a name whose value is at most $m$, and therefore exactly $\ell$ processes return *true*. For linearizability, we partition the operations into two disjoint categories, $C_{true}$ and $C_{false}$, according to their return values. We order all operations in $C_{true}$ before the time that the doorway is set to $closed$, and all operations in $C_{false}$ afterwards. Within each category we order the operations according to the order of non-overlapping operations. It is clear that this order satisfies the sequential specification of the $\ell$-test-and-set object, since all operations that return *true* are linearized before those that return *false*, and there are exactly $\ell$ of those. To show that this order preserves the order of non-overlapping operations, we only need to argue about non-overlapping operations in different categories, since within each category this order is preserved by construction. Let $op_1$ be an operation that returns *true* and $op_2$ be an operation that returns *false* and assume, towards a contradiction, that $op_2$ finishes before $op_1$ starts. Then $op_2$ must set the doorway to *closed*, implying that after $op_1$ reads the doorway it returns *false*. This contradiction concludes the proof that the above implements a linearizable $\ell$-test-and-set object. □

Next, Figure 6.7 shows an implementation of an $\ell$-valued fetch-and-increment object using two smaller fetch-and-increment objects. Note that this recursive construction unfolds to a tree, whose leaves are 0-valued fetch-and-increment objects. We implement such an object with an empty data structure on which the fetch-and-increment operation always returns 0.

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

**Lemma 13.** *If O.left and O.right are linearizable $\ell/2$-fetch-and-increment objects then procedure recursive-fetch-and-increment implements a linearizable $\ell$-fetch-and-increment object.*

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

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

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

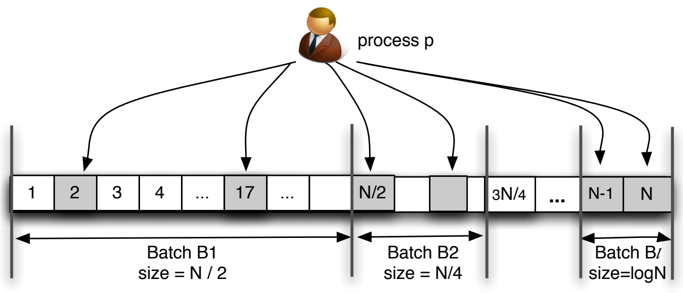

Figure 6.8: The *BitBatching* algorithm. The process makes $\Theta(\log n)$ random trials in each batch, until it first wins a test-and-set object.

**Lemma 14.** *The $\ell$-test-and-set object has expected step complexity $O(\log k)$. The $m$-valued fetch-and-increment object has expected step complexity $O(\log k \log m)$, where $k$ is the number of participating processes.*

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

## 6.5 Non-Adaptive Strong Renaming using Linear Space

In this section, we present an algorithm called *ReShuffle*, which renames into $n$ names, using at most $O(\log^2 n)$ test-and-set operations per process, with high probability. The algorithm assigns unique names to processes by repeatedly sampling over batches of test-and-set bits of decreasing size.

We assume a model in which processes may perform atomic test-and-set operations. Alternatively, the test-and-set operations can be simulated using reads and writes, following the RatRace algorithm in Chapter 5, at the cost of a poly-logarithmic increase in running time. Thus, the time complexity of *ReShuffle* is higher than that of the adaptive renaming network; also, the algorithm is not *adaptive*, since it assumes knowledge of $n$, and will only guarantee names from 1 to $n$ (as opposed to names from 1 to $k$). On the other hand, the *ReShuffle* algorithm uses only $n$ test-and-set objects to assign names, and is simpler than the adaptive sorting network algorithm.

### 6.5.1 Algorithm Description

The $n$ processes share a vector of $n$ test-and-set objects. These objects can either be available in hardware, or implemented using the *RatRace* algorithm. To simplify the presentation, we will assume that the objects are available in hardware, and are atomic. Also, we consider $n = 2^\kappa$, for an integer $\kappa \geq 0$.

We partition the vector of $n$ test-and-set objects in *batches* as follows. Let $\ell = \lfloor \log(n/\log n) \rfloor$. For $1 \leq i < \ell$, batch $B_i$ consists of vector positions from $n(2^{i-1}-1)/2^{i-1}+1$ to $n(2^i-1)/2^i$. In particular, batch $B_1$ consists of the first half of the vector (from left to right), batch $B_2$ consists of the next quarter, and so on. Batch $B_\ell$, which does not follow the above formula, consists of positions from $n(2^{\ell-1}-1)/2^{\ell-1}+1$ to position $n$. For $1 \leq i < \ell$, the length of batch $B_i$ is $n/2^i$. Batch $B_\ell$ has length between $\log n$ and $2\log n$ (see Figure 6.8 for an illustration).

Fix a constant $\beta \geq 6$. Given this partitioning of the vector, processes (sequentially) compete in $\beta \log n$ test-and-set objects in each batch, starting from batch number 1 up to batch $\ell$, stopping when they first win a test-and-set object. More precisely, we define two *stages* in the algorithm. In the first stage, for every $1 \leq i < \ell$, each process $p$ (sequentially) competes in $\beta \log n$ *randomly chosen* test-and-set objects from every batch $B_i$[2]. If the process did not stop before entering the last batch $B_\ell$, the process competes in *every* test-and-set object in this batch. If the process finishes competing in batch $B_\ell$ and still did not win a test-and-set, then it enters the second stage, where it competes in all test-and-set objects from 1 to $n$, in sequence, from left to right. In the following, we will show that, with high probability, every process wins a test-and-set while in the first stage.

### 6.5.2 Algorithm Analysis

The algorithm ensures the same termination guarantees as the underlying test-and-set implementation. In particular, if the test-and-set is provided in hardware, then termination is guaranteed in every execution. Otherwise, if test-and-set is implemented from reads and writes using randomization, the termination property holds with probability 1. The name uniqueness property follows since no two processes may win the same test-and-set. In the following, we prove upper bounds on the step complexity of the algorithm, assuming that the test-and-set operations are provided in hardware and have unit cost.

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

**Lemma 15** (Local Trials)**.** *For $\beta \geq 6$, with high probability in n, every process terminates while executing the first stage, i.e. returns a name after competing in $O(\log^2 n)$ test-and-set objects.*

*Proof.* Assume that there exists a process $p$ that enters the second stage, i.e. $p$ competed in test-and-set objects in all batches ($B_i$) for $i = 1, \ldots, \ell$ without winning any test-and-set object. In particular, this implies that $p$ has competed in all the test-and-set objects in the last batch $B_\ell$. Since $p$ did not win any test-and-set in this batch, it follows that this batch is already "full," i.e. there are at least $\log n$ distinct processes that won each of the test-and-set objects in

[2]A process can pick a random number from a batch by flipping $O(\log n)$ independent fair coins locally.

this batch (we consider the linearization order at each of these objects). Let $S_\ell$ be this set of processes.

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

**Claim 3.** *Fix $\beta \geq 6$, $i \geq 1$, and let $B_i$ be the corresponding batch. Let $S$ be a set of at least $|B_i|/2$ processes that perform $\beta \log n$ random trials each in batch $B_i$, and* none *succeeds in winning a test-and-set in batch $B_i$. Then the batch $B_i$ is* full *with high probability, i.e. there exists a set of processes $F_i$ with $|F_i| = |B_i|$ such that each process in $F_i$ won a distinct test-and-set object in batch $B_i$. Moreover, $S \cap F_i = \emptyset$.*

*Proof.* Given that $|B_i|$ processes performed $\beta \log n$ random trials each in the batch $B_i$, it follows that the batch $B_i$ has been tried by a total of at least $\beta/2 \cdot |B_i| \log n$ random probes. We now use the following probabilistic fact, which is a variant of the well-known *coupon collector* analysis [71, 75].

**Proposition 4** (Coupon Collector)**.** *For $m \geq 1$, consider a set of $m$ bins, initially empty, and a set of $\alpha m \log n$ balls, each thrown independently at random into one of the $m$ bins. The probability that at the end of this process there exists an empty bin is at most $1 - 1/n^{\alpha-1}$, for $c \geq 2$ constant.*

*Proof.* Consider an arbitrary ordering of the $m$ bins, and a sequential ordering of the $\alpha m \log n$ balls. Let $e_i^r$ be the random event that bin number $i \in \{1, \ldots, m\}$ contains no balls after $r \geq 1$ trials have been performed. Then

$$\Pr(e_i^r) = \left(1 - \frac{1}{m}\right)^r.$$

By a standard approximation, we get that $\Pr(e_i^r) \leq e^{-r/m}$. Replacing $r$ with $\alpha m \log n$, we obtain that this probability is upper bounded by

$$\left(\frac{1}{e}\right)^{\frac{\alpha m \log n}{m}} = \left(\frac{1}{e}\right)^{\alpha \log n} = \left(\frac{1}{n}\right)^{\alpha}.$$

By a union bound, we obtain that the probability that there *exists* an empty bin after $\alpha m \log n$ random balls have been thrown is at most

$$\sum_{i=1}^{m} \Pr(e_i^{\alpha m \log n}) = m \left(\frac{1}{n}\right)^{\alpha} \leq n^{-(\alpha-1)}.$$

□

Returning to the proof of the Lemma, we have that the $\beta/2|B_i|\log n$ probes already been scheduled by the adversary. We can use the Proposition above to conclude that *each* of the

$|B_i|$ test-and-sets has been accessed by one of the random probes, with probability at least $1 - 1/n^{\beta/2-1}$. Since we can pick $\beta \geq 6$, it follows that the success probability is at least $1 - 1/n^c$, for some constant $c \geq 2$.

None of these $\beta/2|B_i|\log n$ probes succeeds in winning the test-and-set it accesses, since the processes won test-and-sets in the next batch. By the linearizability of test-and-set, it follows that for each of these test-and-set objects, there exists a *distinct* process that either wins the object or crashes while accessing the object. In both cases, the stops executing probes as part of the algorithm. This holds with probability at least $1 - 1/n^c$, which concludes the proof of the claim. □

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

Using this bound on the number of trials, we obtain bounds on the local and total step complexity of our algorithm.

**Corollary 3.** *With high probability in $n$, every process returns after $O(\log^2 n)$ steps. The expected step complexity of the algorithm is $O(\log^2 n)$. With high probability in $n$, the* total *step complexity of the algorithm is $O(n \log n)$, where we count each test-and-set operation as a step.*

*Proof.* The per-process bound follows trivially from Lemma 15. For the total step complexity bound, we first prove the following claim.

**Claim 4.** *For any $1 \leq i \leq \ell$, there are at most $2\beta|B_i| \cdot \log n$ process probes in batch $B_i$, with high probability.*

*Proof.* Assume for contradiction that there exists a batch in which there are more than $2\beta|B_i|\log n$ trials with non-negligible probability $\gamma$. Since a process performs at most $\beta \log n$ total random trials in a batch (this holds for the last batch $B_\ell$ as well since its length is less than $\beta \log n$), we obtain that there are strictly more than $2|B_i|$ processes that performed random trials in batch $B_i$. Since there are *at most* $2|B_i|$ test-and-set objects in batches $(B_j)_{j\in\{i,i+1,\ldots,\ell\}}$, it follows that, with probability at least $\gamma$, there exists a process $q$ that cannot win a test-and-set object in stage 1. This contradicts Lemma 3. □

Claim 4 ensures that there are at most $2\beta|B_i| \cdot \log n$ process probes in batch $B_i$, with high probability. Summing over all batches, this implies that the total number of test-and-set accesses performed by processes is less than $2\beta n \log n$. □

## 6.6 Related Work

The first paper to analyze randomized renaming in an asynchronous system was by Panconesi et al. [79]. The authors presented a non-adaptive solution ensuring a namespace of size $(1+\epsilon)n$, for $\epsilon > 0$ constant, with expected $O(M \log^2 n)$ total step complexity, where $M$ is the size of the initial namespace. Their strategy was to introduce a new test-and-set implementation, and to award names to processes based on the index of the test-and-set object they acquire. Along the same vein, Eberly et al. [44] obtained *tight* non-adaptive randomized renaming based on the test-and-set by Afek et al. [3]. Their implementation has $O(n \log n)$ amortized step complexity per process, under a given cost measure. The average-case total step complexity of their algorithm is $\Omega(n^3)$.

A paper by Alistarh et al. [9] generalized the approach by Panconesi et al. [79] 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 algorithm with $O(n \text{ polylog } n)$ total step complexity. A modified version of this approach generates an *adaptive* algorithm with similar complexity, which ensures a loose namespace of size $(1+\epsilon)k$, for $\epsilon > 0$ constant.

The algorithms presented in this chapter first appeared in [7]. The renaming network algorithm is the first algorithm to achieve strong adaptive renaming in sub-linear time, improving exponentially on the time complexity of previous solutions. In Chapter 8, we show 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. [24] to follow from earlier results of Aspnes et al. [14]; this is equivalent to our use of sorting networks for non-adaptive renaming in Section 6.2.1.

# Lower Bounds Part III

# 7 The Time Complexity of Deterministic Renaming

In this section, we prove a linear lower bound on the time complexity of deterministic renaming in asynchronous shared memory. The lower bound holds for algorithms using reads, writes, test-and-set, and compare-and-swap operations, and is matched within constants by existing algorithms, as discussed in Section 7.6. 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). Arguably, this covers the spectrum of all “useful" renaming algorithms.

**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 7.1. 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 Chapter 6, the extra complexity cost of this step is an additive factor of $O(\log M(k))$. 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 revert to an $\Omega(k)$ lower bound on the RMR complexity of adaptive mutual exclusion by Anderson and Kim [63]. 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 $k$.

**Applications.** We notice that we can also apply the result to obtain a linear lower bound on the time complexity of *non-adaptive* renaming algorithms, that guarantee names from 1 to some polynomial function in $n$, with $n$ known. We prove this generalization by reduction: we first show that virtually every non-adaptive renaming algorithm can be transformed into a renaming algorithm *with fails*, which returns a *fail* indication whenever the number of processes $k$ accessing the algorithm is $> n$. Then, we show that any renaming algorithm with fails can be used to obtain an *adaptive* renaming algorithm with similar step complexity and

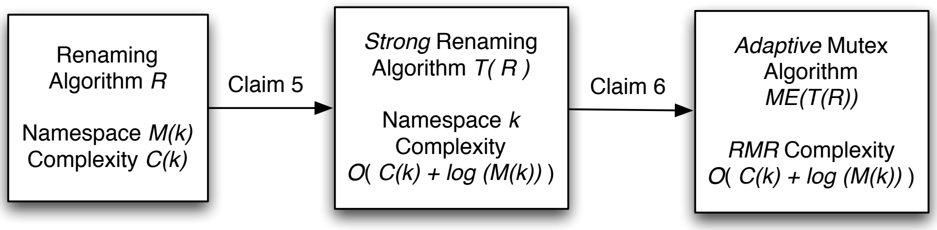

Figure 7.1: The structure of the reduction in Theorem 8.

namespace guarantees. The lower bound follows from our result on adaptive algorithms.

A second application follows from the observation that practical shared-memory objects such as queues, stacks, and fetch-and-increment registers can be used to solve adaptive strong renaming. In turn, this will imply that the linear lower bound will also apply to deterministic shared-memory implementations of these objects using *read*, *write*, *compare-and-swap* or *test-and-set* operations. We analyze the limitations of this lower bound and ways to circumvent it in Section 7.5.

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

In Section 7.1, we give some preliminaries on mutual exclusion and the remote memory reference complexity measure. In Section 7.2 we give the main result, i.e. the linear lower bound on the complexity of deterministic adaptive mutual exclusion. In Section 7.3, we give the generalization to non-adaptive renaming. The applications of the lower bound to practical objects is given in Section 7.4, while Section 7.4.2 presents the mutual exclusion protocol. We conclude with an overview of related work in Section 7.6.

## 7.1 Preliminaries

### 7.1.1 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 a *trying 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 trying 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 critical section.

An algorithm solves mutual exclusion problem with no deadlock if the following hold. We adopt the definition of [26].

- *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 trying section in a configuration, then there is a later configuration in which some process is in the critical section.
- *No lockout*: In every admissible execution, if some process is in the trying 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 thesis, 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, also known as the *contention* in the execution.

### 7.1.2 Remote Memory References (RMRs)

So far, we have used *step complexity*, i.e. the number of shared-memory *read* and *write* operations that a process performs during an execution, to measure the time complexity of an algorithm. In this chapter, we will also employ a stronger measure of complexity, by counting the number of *remote memory references* (RMRs). In cache-coherent (CC) shared memory, each process maintains local copies of shared variables inside its cache. The consistency of the cache among processes is ensured by a coherence protocol. A variable is *remote* to a process if its cache contains a copy of the variable whose value is out of date (or if the cache contains no copy of the variable); otherwise, the variable is *local*. A process step is *local* if it accesses a local variable. Otherwise, the step is a *remote memory reference* (RMR). A similar definition exists for the distributed shared memory (DSM) model. Notice that, since an RMR is a consequence of a process step, for wait-free algorithms the RMR complexity is always a lower bound on step complexity.

### 7.1.3 Queues and Stacks

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

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

- The *Enqueue*($v$) operation changes the state of the stack to $[x_0 = \perp, x_1, \ldots, x_m, v]$, returning *success*.
- The *Dequeue*() operation returns the "oldest" element in the queue. More precisely, if $x_1 \neq \perp$, then the *Dequeue*() operation returns $x_1$ and changes the state of the stack to $[x_0 = \perp, x_2, \ldots, x_m]$. Otherwise, the operation returns $\perp$.

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

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

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

## 7.2 Adaptive Lower Bound

In this section, we prove the following result.

**Theorem 8** (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 $k$ RMRs (and $k$ steps) and (2) each participating process performs a single* rename *operation.*

*Proof.* We begin by assuming for contradiction that there exists a deterministic adaptive algorithm $R$ that renames into a namespace of size $M(k) = 2^{f(k)}$ for $f(k) \in o(k)$, with step complexity $C(k) = o(k)$. The first step in the proof is to show that any such algorithm can be transformed into a wait-free algorithm that solves adaptive *strong* renaming in the same model; the complexity cost of the resulting algorithm will be $O(C(k) + \log M(k))$. This result follows trivially from Theorem 7.

**Claim 5.** *Any wait-free algorithm R that renames into a namespace of size* $M(k)$ *with complexity* $C(k)$ *can be transformed into a tight adaptive renaming algorithm* $T(R)$ *with complexity* $O(C(k) + \log M(k))$.

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.

**Claim 6.** *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 slightly 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 7, each process will eventually be assigned a name from 1 to $k$. Conversely, each name $i$ from 1 to $k$ will eventually get assigned to a unique process $q_i$. Therefore, each of the *Done*[] bits corresponding to names $1,\ldots,k$ will be eventually set to *true*, which implies that eventually each process enters the critical section, as required.
- The *unobstructed exit* condition holds since each process performs a single operation in the exit section.

For the complexity claims, first notice that the RMR complexity of the algorithm $R'$ is at most a constant plus the RMR complexity of algorithm $R$. Next, notice that, once a process obtains the name from algorithm $R$, it performs at most two extra RMRs before entering the critical section, since RMRs may be charged only when first reading the *Done*[$v-1$] register, and when the value of this register is set to *true*. Therefore, the (individual or global) RMR complexity of the mutex algorithm is the same (modulo constant additive factors) as the RMR complexity of the original algorithm $R$. Since the algorithm $R$ is wait-free, its RMR complexity is a lower bound on its step complexity.

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 8, 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.

However, the existence of this algorithm contradicts the $\Omega(k)$ lower bound on the RMR complexity of adaptive mutual exclusion by Anderson and Kim [63, Theorem 2], stated below.

**Theorem 9** (Mutex Time Lower Bound [63])**.** *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 [63].*

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 [63], in which each process performs the entry section once. □

### 7.2.1 Technical Notes

**Relation between $k$ and $n$.** The lower bound of Anderson and Kim [63] from which we obtain our result assumes large values of $n$, the maximum possible number of participating processes, in the order of $k^{2^k}$. Therefore, algorithms that optimize for smaller values of $n$ may be able to circumvent the lower bound for particular combinations of $n$ and $k$. (For example, the lower bound does not preclude an algorithm with running time $O(\min(k, \log n))$ if $n$ is known in advance.) On the other hand, the lower bound applies to all algorithms that work for arbitrary values of $n$.

**Read-write algorithms.** Notice that, although the first reduction step employs *compare-and-swap* (or *test-and-set*) operations for building the renaming network, the lower bound also holds for algorithms that only employ read or write operations, since the renaming network is independent from the original renaming algorithm $R$.

**Progress conditions.** Known adaptive renaming algorithms, e.g. [73, 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 [63] applies to all mutex algorithms ensuring livelock-freedom, and our transformation does not require a strengthening of this progress condition.

## 7.3 Non-Adaptive Renaming Lower Bound

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

### 7.3.1 Renaming with Fails

We define a renaming algorithm *with fails* as a non-adaptive renaming algorithm *RF*, that has the same specification as a renaming algorithm as long as the number of participants $k$ does not exceed the maximum number of participants $N$ that the algorithm allows. On the other hand, if $k > N$, then the algorithm *RF* may return a special value *fail* to the calling process instead of a (unique) name. We call an *instance* of the algorithm a variant of *RF* for some fixed $N$.

**Definition 7** (Renaming with Fails)**.** *The* renaming with fails *task for parameter N in namespace $M(N)$ assumes $k > 0$ processes with unique initial identifiers from an unbounded namespace, and ensures the following.*

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

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

**From non-adaptive renaming to renaming with fails.** In the following, we consider non-adaptive renaming algorithms that ensure the following three properties:

1. Every step of the algorithm executed by a correct process terminates eventually;
2. Any step can only update variables that are part of the algorithm;
3. The algorithm works for unbounded input namespace size.

It is straightforward to see that any wait-free algorithm can be modified to ensure properties (1) and (2), while property (3) is ensured by the original specification of the (non-adaptive) renaming problem, given in [17]. Given these properties, we give a procedure to transform any non-adaptive renaming algorithm $R$ into a renaming algorithm with fails $RF$ with the same asymptotic complexity.

Let $M(N)$ be the namespace that $R$ renames into, and let $C(N)$ be $R$'s complexity. We associate with each process a local program counter that counts the number of steps that the process has performed in the current execution. Processes share an instance of the algorithm $R$, and an array *Split* of $M(N)$ deterministic splitter objects, as defined in Chapter 5.

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

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

**Lemma 16** (Renaming with Fails)**.** *For any $N > 0$, given a renaming algorithm $R$ for at most $N$ processes with complexity $C(N)$ ensuring a namespace of size $M(N)$, the transformation yields a renaming with fails algorithm $RF$ with parameter $N$, having the same asymptotic step complexity as $R$, ensuring the same properties as $R$ as long as $k \leq N$.*

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

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

### 7.3.2 From Renaming with Fails to Adaptive Renaming

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

**Transformation.** We start from a non-adaptive renaming with fails algorithm $R$, which, for any $N \geq 1$, renames into a namespace of size $M(N)$, with complexity $C(N)$, as long as the number of participants $k$ to the instance does not exceed $N$. We consider an infinite series $(R_i)_{i=1,2,\ldots}$ of instances of algorithm $R$, where instance $R_i$ is the algorithm $R$ for parameter $N = 2^i$.

The transformation proceeds as follows. Each process accesses the instances $(R_i)_i$ in order, until it first obtains an integer name from an instance $R_i$ (as opposed to a *fail* indication). If, on the other hand, a process obtains a *fail* indication from $R_i$, it increments its instance counter $i$, and proceeds to the next instance. Once it has obtained a name $v$, the process returns $v$ plus the sum resulting from adding up the namespace sizes for the previous instances, i.e., for $j \geq 2$, $\sum_{j=1}^{i-1} M(2^j)$.

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

**Lemma 17.** *Let $A$ be an algorithm that renames with fails such that, for any $N \geq 1$, it guarantees a namespace of size polynomial in $N$ with step complexity $o(N)$. Then the above transformation yields an* adaptive *renaming algorithm that renames in a namespace polynomial in the number of participants $k$, whose complexity is $o(k)$.*

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

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

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

Finally, we bound the step complexity of the transformation. By its structure, the number of steps a process takes in total is bounded by $C(2^m) + C(2^{m-1}) + \ldots + C(1)$. Since $R_m$ is the highest accessed instance of algorithm $R$, it follows that $2^m > k \geq 2^{m-1}$. We want to show that $C(2^m) + C(2^{m-1}) + \ldots + C(1)$ is in $o(2^m) = o(k)$, knowing that $C(2^m) = o(2^m)$. Therefore, we need to show that the quantity

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

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

**Lemma 18** (Stolz-Cesàro)**.** *Let $(A_n)_{n\geq 1}$ and $(B_n)_{n\geq 1}$ be two sequences of real numbers, such that $(B_n)_{n\geq 1}$ is positive, strictly increasing, and unbounded. Then*

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

*if the limit on the right hand side exists.*

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

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

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

On the other hand, the existence of such an adaptive renaming algorithm contradicts Theorem 8. Therefore, it follows that every deterministic renaming algorithm with fails with parameter $N$, guaranteeing a namespace polynomial in $N$ has complexity $\Omega(N)$. From Lemma 16, the same result holds for non-adaptive renaming algorithms.

**Corollary 4.** *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 $\Omega(n)$.*

## 7.4 Applications

### 7.4.1 Lower Bounds for Other Objects

These results imply time lower bounds for implementations of other shared objects, such as fetch-and-increment registers, queues, and stacks. Some of these results are new, while others improve on previously known results.

We first show reductions between fetch-and-increment, queues, and stacks, on the one hand, and adaptive strong renaming, on the other hand.

**Lemma 19.** *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 by initializing it with $n$ distinct integers $1, 2, \ldots, n$, where 1 is the element at the head of the queue, and $n$ is the element at the tail 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. Again, correctness follows trivially from the sequential specification of the queue.

Finally, given a stack, we initialize it with values $1, 2, \ldots, n$, where 1 is the top of the stack. To solve renaming, each process performs *pop* on the stack and returns the element received. □

This implies a linear time lower bound for these objects.

**Corollary 5** (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).*

### 7.4.2 A Time-Optimal Non-Adaptive Mutex Algorithm

Another application of the lower bound argument is that we can obtain an asymptotically optimal mutual exclusion algorithm from an AKS sorting network [5]. We present this algorithm in the cache-coherent (CC) model.

**Description.** Processes share an AKS sorting network with $n$ input (and output) ports, and

```
Shared:
An AKS renaming network $R$, with $n$ input and output ports
An array Done of $n$ registers, initially false
procedure entry-section($v_i$)
  $w \leftarrow$ input wire corresponding to $v_i$
  while $w$ is not an output wire do
    $T \leftarrow$ next test-and-set on wire $w$
    $res \leftarrow T$.test-and-set()
    if $res = 0$ then
      $w \leftarrow$ output wire $x'$ of $T$
    else
      $w \leftarrow$ output wire $y'$ of $T$
  /* $w$ is an output wire */
  $id_i \leftarrow w.index$
  if $id_i = 1$ then
    execute critical section
  else
    spin until $Done[id_i - 1] = true$
    execute critical section
  return
procedure exit-section()
  $Done[id_i] \leftarrow true$
  return
```

Figure 7.2: Pseudocode for the mutex algorithm.

a vector *Done* of boolean bits, initially *false*. We replace each comparator in the network with a two-process test-and-set object with constant RMR complexity [52, 53]. In the mutual exclusion problem processes are assumed hold unique initial identifiers $v_i$ from 1 to $n$, therefore we use these initial identifiers to assign unique input ports to processes. A process progresses through the network starting at its input port, competing in test-and-set objects, as in a renaming network. A process takes the top comparator output if it wins (returns 0 from) the test-and-set, and the bottom output otherwise. The process adopts the index of the output port it reaches as a temporary name $id$. If $id = 1$, then it enters the critical section; otherwise it busy-waits until the bit $Done[id-1]$ is set to *true*. Upon exiting the critical section, the process sets the $Done[id]$ bit to true.

The correctness of the algorithm above follows from Claims 5 and 6. In particular, the asymptotic local RMR complexity of the above algorithm is the same as the depth of the AKS sorting network (plus at most two RMRs for reading the *Done* bits), i.e. $O(\log n)$, therefore the algorithm is optimal by the lower bound of Attiya et al. [23]. Anderson and Yang [54] presented an upper bound with the same asymptotic complexity, but significantly better constants, using a different technique. The same construction can be used starting from constuctible sorting networks, e.g. bitonic sorting networks [64], at the cost of increasing complexity by a logarithmic factor. Notice that the linear RMR lower bound of Anderson and Kim [63] does not yield a $\Omega(n)$ RMR lower bound for *non-adaptive* mutual exclusion (which would contradict the existence of our algorithm).

## 7.5 Circumventing the Lower Bound

In brief, the lower bound shows that, if the number $n$ of potential participating processes is very large, then, for any $k \ll n$, one can obtain a schedule with participating processes in which one process takes a step for roughly every other participating process. There are two ways for algorithms to circumvent the argument.

**Randomization.** As shown in Chapter 6, one can avoid the worst-case linear schedules with high probability by allowing processes to flip coins as part of the execution. In particular, for renaming, the complexity of an operation is $O(\log k)$ with high probability, i.e. exponentially less than the worst-case deterministic schedule.

**Limiting the parameter $n$ or the size of the initial namespace.** Another way of circumventing the lower bound is assuming that the maximum number of participants $n$ is small, and that processes already have unique names from 1 to $n$. (For example, this can be achieved by running a renaming algorithm at the beginning of the execution.) An example of such an algorithm is the $O(\log n \log v)$ counter algorithm by Aspnes et al. [12], which assigns a unique "input port" in the data structure to each of the $n$ processes. For $n$ large, if $k = \Theta(\log n)$ processes are participating, each of them performs a number of operations which is at least linear in $k$, although this number is in $O(\log n \log v)$. For $n$ small, linear contention in $k$ can be seen as negligible.

## 7.6 Related Work

Renaming was introduced in [17], where the authors showed a lower bound of $(n+1)$ names in the wait-free asynchronous case. The lower bound on namespace size for deterministic read-write solutions was improved to $(2n-2)$ in a landmark paper by Herlihy and Shavit [57], with refinements by Rajsbaum and Castañeda [34, 35]. This lower bound can be circumvented using hardware compare-and-swap or test-and-set operations [73], as well as using randomization [44] (at the cost of allowing a vanishing probability that the algorithm does not terminate). Renaming has been shown to be related to weak symmetry breaking in [50]; it is also related to the processor identity problem [65]; the key difference is that, for renaming, participants are assumed to have distinct initial identifiers (from an unbounded namespace).

Several renaming algorithms were proposed in the literature, e.g. [17, 31, 73, 21, 9, 7]. The lower bound given in this chapter is matched by the algorithm of [73] which assumes hardware test-and-set operations; for read-write algorithms, it is matched by the algorithm of [73], which achieves a namespace of size $O(k^2)$; it is also matched (within a logarithmic factor) by the algorithm of [21], which ensures a linear namespace of size $(6k-1)$. Chlebus and Kowalski [39] showed a linear lower bound for deterministic renaming algorithms under the assumption that the number of available registers is limited. (By contrast, our lower bounds do not require such a restriction.)

As we pointed out, the lower bound applies to fetch-and-increment, queues, and stacks, and extends previous results obtained for these objects. Indeed, Jayanti, Tan and Toueg [61], as well as Ellen et al. [48] already presented linear lower bounds for deterministic counters, queues and stacks. One limitation of these two results is that the worst-case executions they build require processes to perform an exponential number of operations–by contrast, there exist counter implementations that have polylogarithmic step complexity for polynomially many increment operations [12]. Our linear bound does not have this limitation, since each process performs only one operation in the worst-case execution (note that our deterministic linear lower bound does not apply to counters). In essence, we show that the linear threshold is inherent for worst-case executions of fetch-and-increment, queues, and stacks, even if each process performs only one operation. For practical objects, the linear lower bound can be matched by universal construction implementations, which use reads, writes, and compare-and-swap operations [56, 45].

# 8 A Tight Time Lower Bound for Adaptive Randomized Renaming

In this section, we present lower bounds on the expected *total* step complexity of randomized renaming and counting. (Naturally, the result also applies to deterministic algorithms, yielding a worst-case total step complexity lower bound.) The lower bound holds for algorithms using reads, writes, test-and-set, or compare-and-swap operations, and is matched by the renaming network algorithm, as discussed in Section 8.7.

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

Our argument generalizes a previous result by Jayanti [60], which in turn is similar to a lower bound by Cook, Dwork, and Reischuk [40] on the complexity of computing basic logical operations on PRAM machines. Jayanti proved an $\Omega(\log k)$ lower bound on the expected step complexity of shared counters, queues, and stacks, which can be tweaked to apply to strong adaptive renaming. We generalize his result in two ways: first, we consider *total* step complexity, and thus obtain a stronger $\Omega(\log k)$ lower bound on the *average* worst-case expected step complexity of the problem. Second, our results also apply to loose (approximate) versions of renaming and counting, showing that the time complexity benefits of relaxing the object semantics in this way are at most constant.

We begin by discussing some basic definitions and properties related to adversarial schedules in Section 8.1. We then describe the adversarial scheduler and analyze its properties in Section 8.2. From these properties, we obtain the lower bound for adaptive renaming in Section 8.3. We refine this argument to obtain a lower bound for approximate counting in Section 8.4. We then discuss applications to more complex objects 8.6 and conclude with an overview of related work in Section 8.7.

## 8.1 Preliminaries

### 8.1.1 Loose Adaptive Renaming and Approximate Counting

We recall the definitions of these two objects. For $c \geq 1$, the $c$-loose renaming problem requires processes to return unique names from 1 to $ck$, where $k$ is the contention in the execution.

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

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

where $\gamma$ is the number of increment operations linearized before the read operation $R$.

### 8.1.2 Operations and Schedules

Recall that we consider a shared-memory model with atomic read, write, and compare-and-swap operations. Notice that the read and write operations always succeed, whereas the compare-and-swap operation is *conditional*, meaning that it may or may not change the value of the register on which it is called, depending on the value of the register when the operation is called. Notice that certain operations change the value of the underlying object (such as a write with a new value), whereas others are “invisible", such as reads, and failed compare-and-swap operations. In the following, we make this intuition precise, and analyze the existence of schedules in which few operations are visible. We follow the presentation of [22], where these notions were first defined.

**Definition 8** (Invisible Operations [22])**.** *Let $e$ be an operation applied by a process $p$ to an object $O$, in an execution $E = E_1 e E_2$. We say that $e$ is* invisible *in $e$ if either the value of the object $O$ is not changed by $e$, or if $E_2 = E' e' E''$, where $e'$ is a write operation on $O$, $E'$ contains no operations by $p$, and no operation in $E'$ is applied to $O$. If $e$ is not invisible in $E$, we say that it is* visible *in $E$.*

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

**Definition 9** (Weakly-Visible Schedule [22])**.** *Let $S = \{e_1, \ldots, e_\ell\}$ be a set of operations by different processes that are enabled after some execution prefix $E$, all about to apply write or compare-and-swap operations. We say that an ordering of the operations in $S$ is a* weakly-visible *schedule of $S$ after $E$, denoted by $\sigma(E, S)$, if the following holds. Let $E_1 = E\sigma(E, S)$.*

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

Given these definitions, Attiya and Hendler [22] proved the following result on the existence of weakly-visible schedules. The proof follows by constructing a suitable ordering of the operations in $S$.

**Lemma 20** (Weakly-Visible Schedules [22]). *Let $S = \{e_1, \ldots, e_\ell\}$ be a set of operations by different processes that are enabled after some execution $E$, all about to apply write or compare-and-swap operations. Then there is a weakly-visible-schedule of $S$ after $E$.*

### 8.1.3 Worst-case Expected Step Complexity

In the following lower bounds, we will use the following basic fact, whose proof follows from the definition of the expectation of a random variable.

**Proposition 5** (Expected Complexity). *Fix constants $\alpha \in [0,1]$ and $\gamma > 0$. Given an algorithm $A$ that terminates with probability $\alpha$, if there exists an adversarial strategy $\mathscr{S}(A)$ such that, in every execution under $\mathscr{S}(A)$ in which every process terminates, the processes take at least $\gamma$ steps, then the (worst-case) expected step complexity of $A$ is at least $\alpha\gamma$.*

## 8.2 The Adversarial Scheduler

We consider an algorithm $A$ in shared-memory allowing atomic *read*, *write*, and *compare-and-swap* operations (notice that the *test-and-set* operation can be trivially replaced by *compare-and-swap*. The adaptive adversary follows the pseudocode described in Figure 8.1. The adversary schedules the processes in rounds: in each round, each process that has not yet returned from $A$ is scheduled to perform the next shared-memory operation that they have enabled. More precisely, at the beginning of each round, the adversary allows each process to perform local coin flips until it either terminates or has to perform an operation that is either a *read*, a *write*, or a *compare-and-swap* (lines 3-5).

In each round, the adversary partitions processes that have an operation enabled into three sets: $\mathscr{R}$, the *readers*, $\mathscr{W}$, the *writers*, and $\mathscr{C}$, the *swappers*. Processes in $\mathscr{R}$ are scheduled by the adversary to perform their enabled *read* operations, in the order of their initial identifiers (line 9). Then each process in $\mathscr{W}$ is scheduled to perform the *write*, again in the order of initial identifiers (line 10). Finally, the swappers are scheduled following a particular *weakly-visible* schedule $\sigma$, formally defined and shown to exist in Lemma 20, whose goal is to minimize the information flow between processes. Once each process has either been scheduled or has returned, the adversary moves on to the next round.

Before we proceed with the analysis, we explain the role of the weakly-visible schedule for the processes performing compare-and-swap operations in round $r$. Notice that, if a set of processes all perform compare-and-swap operations in a round, there exist interleavings of these operations such that the last scheduled process “finds out" about *all* other processes after performing its *compare-and-swap*, by reading a value that these processes successively

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

Figure 8.1: The adversarial strategy for the global lower bound.

modified. However, the adversary can always break such interleavings and ensure that, given any set of *compare-and-swap* operations, a process only finds out about *one* other operation, using a *weakly-visible* schedule, as described in Section 8.1.2.

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

We prove that the adversarial strategy described above prevents any process from "finding out" about more than $2^r$ active processes by the end of round $r$ in any execution from $\mathscr{S}(A)$. More precisely, for each process $p$ following the algorithm $A$, each register $R$, and for every round $r \geq 0$, we define the sets $UP(p,r)$ and $UP(R,r)$, respectively. Intuitively, $UP(p,r)$ is the set of processes that process $p$ might know at the end of round $r$ as having taken a step in an execution resulting from the adversarial strategy. Similarly, $UP(R,r)$ is the set of processes that can be inferred to have taken a step in an execution resulting from the adversarial strategy, by reading the register $R$ at the end of round $r$. Our notation follows the one in [60], which defines similar measures for a model in which LL/SC, *move*, and *swap* operations are available.

Formally, we define these sets inductively, using the following update rules. Initially, for $r = 0$, we consider that $UP(p,0) = \{p\}$ and $UP(R,0) = \emptyset$, for all processes $p$ and registers $R$. For any later round $r \geq 1$, we define the following update rules:

1. At the beginning of round $r \geq 1$, for each process $p$ and register $R$, we set $UP(p,r) =$

$UP(p, r-1)$ and $UP(R, r) = UP(R, r-1)$;

2. If process $p$ performs a successful *write* operation on register $R$ in round $r$, then $UP(R, r) = UP(p, r-1)$. Informally, the knowledge that process $p$ had at the end of round $r-1$ is reflected in the contents of register $R$ at the end of round $r$. On the other hand, the writing process $p$ gains no new knowledge from writing, i.e. $UP(p, r) = UP(p, r-1)$;

3. If process $p$ performs a successful *compare-and-swap* operation on register $R$ in round $r$, i.e. if the operation returns the expected value, then the information contained in the register is overwritten with $p$'s information, that is $UP(R, r) = UP(p, r-1)$. We also assume that the process $p$ gets the information previously contained in the register $UP(p, r) = UP(p, r-1) \cup UP(R, r)$ (the contents of $UP(R, r)$ might have been already updated in round $r$);

4. If process $p$ performs an unsuccessful *compare-and-swap* operation on register $R$ in round $R$, then $UP(R, r)$ remains unchanged. On the other hand, the process gets the information currently contained in the register, i.e. $UP(p, r) = UP(p, r-1) \cup UP(R, r)$;

5. If process $p$ performs a successful *read* operation on register $R$ in round $r$, then $UP(R, r)$ remains unchanged, and $UP(p, r) = UP(R, r) \cup UP(p, r-1)$.

Based on these update rules, we can compute an upper bound on the size of the *UP* sets for processes and registers, as the rounds progress.

**Lemma 21** (Bounding information)**.** *Given a run of the algorithm A controlled by the adversarial scheduler in Figure 8.1, for any round $r \geq 0$, and for every process or shared register $X$, $|UP(X, r)| \leq 2^r$, where the set UP is considered at the beginning of the round $r$.*

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

Given $r > 0$ for which the claim holds, we prove it for $r + 1$. We first prove the claim for $UP(R, r+1)$, where $R$ is a register. Notice that the amount of information in $R$ may change only as a consequence of it being written, or updated by a compare-and-swap operation. In both cases, the set $UP(R, r+1)$ takes the value of $UP(p, r)$. By the induction step, this is at most $4^r$, and the claim holds.

We now consider $UP(p, r+1)$ for a process $p$. Following the update rules, since the adversary uses a weakly-visible schedule, the size of the set $UP(p, r+1)$ can be at most

$$|UP(p, r)| + |UP(q, r)|,$$

where $q$ another process, whose operation is visible to $p$ during round $r$. The claim follows by the induction step. □

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

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

Starting from the execution $\mathscr{E}$, the adversary can build an execution $\mathscr{E}'$ in which only processes in $S$ participate, that is indistinguishable from $\mathscr{E}$ from $p$'s point of view, by starting from execution $\mathscr{E}$ and only scheduling processes in $S = UP(p, r)$ up to the end of round $r$ of $\mathscr{E}'$. The proof is identical to the one presented by Jayanti [60] in the context of local lower bounds for shared-memory with LL/SC operations. Therefore, we only give an overview of the construction in this thesis.

**Lemma 22** (Indistinguishability)**.** *Let $\mathscr{E}$ be an execution in $\mathscr{S}(A)$ and $p$ be a process with $UP(p, r) = S$ at the end of round $r$. There exists an execution $\mathscr{E}'$ of $A$ in which only processes in $S$ take steps, such that $\mathscr{E}$ and $\mathscr{E}'$ are* indistinguishable *to process $p$.*

*Proof.* To prove the claim, we provide an algorithm for the adversary to build an execution $\mathscr{E}'$ based on the original execution $\mathscr{E} \in \mathscr{S}(A)$, and prove that $\mathscr{E}'$ and $\mathscr{E}$ are indistinguishable for process $p$ at the end of round $r$. The construction is an adaptation to the read-write model of the one presented by Jayanti in [60]. Given the execution $\mathscr{E}$, let $coins(\mathscr{E}, p, j)$ be the outcome of the $j$th coin toss that process $p$ performs in execution $\mathscr{E}$.

**Constructing the execution.** The procedure to build the desired execution $\mathscr{E}'$ of algorithm $A$ in which only $|S|$ processes participate is described in Algorithm 8.2. The run is also structured in rounds. Of note, only processes that are scheduled in round $r$ are the processes in $S$ that have not witnessed processes outside of $S$ by the end of round $r - 1$. Each process is scheduled to perform local coin tosses until it has a shared-memory operation enabled. For every coin toss $j$ by a process $q$, the adversary feeds the outcome that occurred in the execution $\mathscr{E}$, that is $coins(\mathscr{E}, q, j)$. Depending on their enabled shared-memory operation, the processes that have not yet terminated are then split into a set of readers, a set of writers, and a set of swappers, that is processes having compare-and-swap operations enabled. The readers are then scheduled in the order of their initial identifiers, after which the writers are scheduled in the order of their initial identifiers. The swappers are scheduled in the order of their weakly visible schedule. Finally, the adversary increments the round counter and moves to the next round.

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

Figure 8.2: The procedure for building the indistinguishable execution.

**Correctness of the construction.** The proof of correctness proceeds by induction on the round number $r$, and is identical to the one outlined in [60], Lemma 5.2. More precisely, the execution $\mathscr{E}$ is the $(All, \mathscr{A})$ run, and the execution $\mathscr{E}'$ is the $(S, \mathscr{A})$-run. We refer the reader to reference [60] for the proof.

□

## 8.3 Renaming Lower Bound

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

**Theorem 10** (Renaming)**.** *Fix $c \geq 1$ constant. Given $k$ participating processes, any $c$-loose adaptive renaming algorithm that terminates with probability $\alpha$ has worst-case expected total step complexity $\Omega(\alpha k \log(k/c))$.*

*Proof.* Let $A$ be a $c$-loose adaptive renaming algorithm. We consider a *terminating* execution $\mathcal{E} \in \mathcal{S}(A)$ with $k$ participating processes, i.e. every participating process returns in $\mathcal{E}$. We first prove that a process that returns name $j \in [1, ck]$ in execution $\mathcal{E}$ has to perform $\Omega(\log(j/c))$ shared-memory operations.

First, notice that each execution $\mathcal{E} \in \mathcal{S}(A)$ contains no process failures, so each process has to return a unique name in the interval $1, \ldots, ck$ in such an execution. Therefore, there exist distinct names $m_1, \ldots, m_k \in \{1, 2, \ldots, ck\}$ and processes $q_1, \ldots, q_k$ such that process $q_i$ returns name $m_i$ in execution $\mathcal{E}$. Without loss of generality, assume that the names returned by processes $q_1, \ldots, q_k$ are in monotonically increasing order; since the names are distinct, we have that $m_i \geq i$ for $i \in 1, \ldots, k$.

Consider process $q_i$ returning name $m_i$ in $\mathcal{E}$. Let $\ell_i$ be the number of shared-memory operations that $q_i$ has performed in $\mathcal{E}$. Since the adversary schedules each process once in every round of $\mathcal{E}$, until termination, it follows that process $q_i$ has returned at the end of round $\ell_i$. Let $S = UP(q_i, \ell_i)$, as defined in Section 8.2. Since $\mathcal{E} \in \mathcal{S}(A)$, by Lemma 21, we have that $|S| \leq 2^{\ell_i}$.

Assume for the sake of contradiction that the number of processes that $q_i$ "found out" about in this execution, $|S|$, is less than $m_i/c$. By Lemma 22, there exists an execution $\mathcal{E}'$ of $A$ which is indistinguishable from $\mathcal{E}$ from $q_i$'s point of view at the end of round $\ell_i$, in which only $|S| < m_i/c$ processes take steps. However, since the algorithm is $c$-loose, the highest name that process $q_i$ may return in execution $\mathcal{E}'$, and thus in $\mathcal{E}$, is *strictly less* than $c \cdot (m_i/c) = m_i$, a contradiction.

Therefore, it has to hold that $|S| \geq m_i/c$, which implies that $\ell_i$, the number of shared-memory operations that process $q_i$ performs in $\mathcal{E}$, is at least $\log_2(m_i/c) = \log(m_i/c)$. Therefore, for any $i \in 1, \ldots, k$, process $q_i$ returning name $m_i$ has to perform at least $\log(m_i/c)$ shared memory operations. Then the total number of steps that the $k$ processes perform in execution $\mathcal{E}$ is

$$\sum_{i=1}^{k} \ell_i \geq \sum_{i=1}^{k} \log(i/c) = \Omega(k \log(k/c)),$$

where in the last step we have used the standard Stirling approximation of $k!$. Since this complexity lower bound holds for every execution resulting from the adversarial strategy, using Proposition 5, we obtain that the expected total step complexity of the algorithm $A$ is $\Omega(\alpha k \log(k/c))$. □

## 8.4 Counting Lower Bound

Using a similar argument, we can show that any $c$-approximate counter implementation has worst-case expected total step complexity $\Omega(k \log(k/c^2))$ in executions where each process performs one increment and one read.

One key difference from the proof in the previous section, which implies the extra $c$ factor, is that processes may return the same value from the *read* operation; we take this into account by studying the linearization order of the increment operations.

**Theorem 11** (Counting)**.** *Fix* $c \geq 1$ *constant. Let* $A$ *be a linearizable* $c$*-approximate counter implementation that terminates with probability* $\alpha$*. For any* $k$*, the algorithm* $A$ *has worst-case expected total step complexity* $\Omega(\alpha k \log(k/c^2))$*, in runs where each process performs an* increment *followed by a* read *operation.*

*Proof.* Let $A$ be a $c$-approximate counting algorithm in this model. We consider *terminating* executions $\mathscr{E}$ with $k$ participating processes, in which each process performs an *increment* operation followed by a *read* operation, during which which the adversary applies the strategy described in Figure 8.1, i.e. $\mathscr{E} \in \mathscr{S}(A)$.

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

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

Second, consider process $p_i$ returning value $v_i$ in execution $\mathscr{E}$. Let $\ell_i$ be the number of shared-memory operations that $p_i$ has performed in $\mathscr{E}$. Since the adversary schedules each process once in every round of $\mathscr{E}$, it follows that process $p_i$ has returned at the end of round $\ell_i$. Let $S = UP(p_i, \ell_i)$, as defined in Section 8.2. By Lemma 21, we have that $|S| \leq 2^{\ell_i}$.

Assume for the sake of contradiction that $|S| < v_i / c$. By Lemma 22, there exists an execution $\mathscr{E}'$ of $A$ which is indistinguishable from $\mathscr{E}$ from $q_i$'s point of view at the end of round $\ell_i$, in which only $|S| < v_i / c$ processes take steps. However, since the counter is $c$-approximate, the highest value that process $p_i$ can return in execution $\mathscr{E}'$, and thus in $\mathscr{E}$, is *strictly less* than $c \cdot (v_i / c) = v_i$, a contradiction.

Therefore, $|S| \geq v_i / c$, and $\ell_i \geq \log(v_i / c) \geq \log(i/c^2)$, for every $1 \leq i \leq k$. We obtain that the total number of steps is bounded as follows.

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

□

## 8.5 Circumventing the Lower Bound

In brief, the lower bound in this chapter states that for implementations of adaptive renaming and related counting objects, the average worst-case expected step complexity is *logarithmic* if the adversary is adaptive, i.e. may adapt its schedule based on the results of the processes' coin flips. Thus, this logarithmic threshold is stronger than the linear one shown in Chapter 7, as it cannot be avoided by the use of randomization in the presence of a strong adversary.

On the other hand, if the adversary is weak, or *oblivious*, and may not adapt the schedule based on the results of the processes coin flips (i.e. fixes the schedule before the execution), then there exist objet implementations that avoid this lower bound. In particular, the approximate counter of Bender and Gilbert [29] guarantees a constant approximation factor with expected running time $O(\log\log n)$ against an oblivious adversary.

A third method of potentially circumventing the lower bound would be to allow the algorithm to break the approximation guarantee with small probability. Notice that the lower bound argument, as written, only applies to algorithms that guarantee approximation within a factor of $c$ in *all* executions. The counter of Bender and Gilbert [29] is an example of such an object, since it guarantees the constant approximation only with high probability. Another example is the *AdaptiveSearch* renaming algorithm of [9], which only ensures a namespace from 1 to $ck$ with high probability in $k$.

## 8.6 Applications to Other Objects

Given that adaptive renaming can easily be solved using queues, stacks, or fetch-and-increment objects, as shown in Section 7.4, the lower bound in Section 8.3 applies to these objects.

**Corollary 6** (Applications)**.** *Consider a wait-free linearizable (randomized) implementation $A$ of a fetch-and-increment register, queue, or stack, in shared memory with read, write, test-and-set and compare-and-swap operations. Then, for any $k \geq 1$, if $A$ terminates with probability $\alpha$, then its expected worst-case step complexity is $\Omega(\alpha k \log k)$, where $k$ is the number of participating processes.*

## 8.7 Related Work

The first lower bound to apply to randomized renaming was a logarithmic lower bound on the individual step complexity of *strong* renaming, derived in [7], generalizing the technique of Jayanti [60]. The lower bound in this chapter was first published in [8], and improves on Jayanti's result [60] and the subsequent generalization by Alistarh et al. [7] in two ways: first, since we consider *total* step complexity, our results imply the local bounds of [60, 7]; second, the results in this chapter also cover the *loose* version of the problem, which relaxes the tight namespace requirements. The lower bound technique by Jayanti, which we generalize, is in turn similar to a lower bound by Cook, Dwork, and Reischuk [40] on the complexity of computing basic logical operations on PRAM machines.

A similar technique was used by Attiya et al. [22], who analyzed the complexity of deterministic implementations when stronger $\ell$-location compare-and-swap primitives are available in memory. They derived $\Omega(n \log_\ell n)$ time lower bounds on deterministic data structures implementing counters, queues, or stacks. The results in this chapter are stronger in that they apply to randomized and approximate versions of these objects, although, as given, they do not apply to the case when $\ell$-location compare-and-swap is available.

The $\Omega(k \log(k/c))$ renaming lower bound is matched asymptotically for $c = 1$ by the renaming algorithm presented in Chapter 6. For counters, the deterministic algorithm of Aspnes et al. [12] guarantees exact counting with complexity $O(\log n \log m)$, where $m$ is the maximal value of the counter, and thus matches the lower bound within a logarithmic factor for $c = 1$ and $k = n = m$. Since this almost-matching algorithm [12] is deterministic and exact, this lower bound limits the gain that can be obtained by randomization or approximation to a logarithmic factor in a shared-memory setting. Note that the counter implementation given in Section 6.4.1 does not match this lower bound, since it is not linearizable.

# 9 Summary and Open Questions

This thesis studies the complexity of concurrent data structures in shared memory, motivated by the classic renaming problem. We have presented and analyzed randomized algorithms for renaming, test-and-set, and counters, and we gave tight lower bounds for deterministic and randomized renaming. Despite this progress, open questions remain. We enumerate some of these questions in this chapter.

## 9.1 Renaming

In Chapter 7, we have given a linear lower bound on the step complexity of deterministic adaptive renaming into a sub-exponential namespace. We also extended this result to a linear lower bound on the step complexity of non-adaptive renaming into a polynomial namespace.

These bounds are matched by algorithms in the literature: for algorithms assuming hardware test-and-set primitives, the matching upper bounds achieve a tight namespace [72]. For algorithms using only reads and writes, which have been studied more extensively in the literature, the algorithm of Chlebus and Kowalski [39] matches the time lower bound, giving a namespace of size $(8k - \log k - 1)$; an elegant algorithm by Attiya and Fouren [21] 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. [4], with time complexity $O(k^2)$. Thus, obtaining a read-write deterministic algorithm which is optimal both in terms of time complexity *and* namespace size is an intriguing open question.

In Chapter 6, we presented a randomized renaming algorithm with logarithmic complexity with high probability, which achieves a tight adaptive namespace. The total time complexity lower bound in Chapter 8 shows that this algorithm is optimal against an adaptive adversary, and that no asymptotic time complexity gains may be obtained for adaptive renaming by relaxing the namespace size within constant factors. This implies that our randomized solution is optimal from two points of view: time complexity *and* namespace size.

One disadvantage of the renaming network algorithm is that it is based on an AKS sorting network [5], which has prohibitively high constants hidden inside the asymptotic notation [64]. 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.

The global lower bound applies to deterministic adaptive algorithms as well, implying an $\Omega(k \log k)$ worst-case global time complexity lower bound. On the other hand, the fastest known algorithm to rename into a namespace of size $ck$ for $c \geq 1$ constant has $O(k^2)$ total step complexity [39]. Closing the gap between upper and lower bounds is still an open question.

Another open question concerns randomized shared-memory renaming in weaker adversarial models, in particular in the *oblivious* adversary model, i.e. when the scheduler has knowledge of the algorithm but fixes the schedule without knowing the results of random coin flips. In this case, there are indications that the logarithmic threshold could be broken, since sub-logarithmic algorithms have been shown to exist for test-and-set [6] and approximate counters [29] against the oblivious adversary.

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. Our renaming network algorithm uses $O(k \log k)$ registers assuming hardware test-and-set operations, while the *BitBatching* implementation uses $O(n)$ registers. The linear space complexity lower bounds of Jayanti, Tan, and Toueg [61] apply to adaptive renaming, therefore the renaming network is space-optimal within a logarithmic factor. However, the tight space complexity thresholds for renaming remain an open question.

## 9.2 Counting and Set Data Structures

The individual (per-process) and total step complexity lower bounds in Chapters 7 and 8 also extend to counting and set data structures. In particular, the individual lower bound suggests that deterministic implementations of data structures solving renaming have worst-case schedules with *linear* time complexity. Practically, this implies that such widely-used data structures do not scale in the worst case, so a key question is how to circumvent this lower bound.

One natural alternative, which we emphasize in this thesis, is the use of *randomization* in the design of concurrent data structures. In fact, we show that, using randomization, this lower bound can be broken in the case of renaming, as we exhibit randomized algorithms with exponentially better step complexity, with high probability. Thus, it is very tempting to ask whether these randomized techniques can be extended to obtain fast sub-linear versions of practical concurrent data structures.

On the other hand, 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 [6, 29] suggest that weaker adversarial models and relaxing object semantics, e.g. allowing approximate implementations, could be used to go below this logarithmic threshold.

## 9.3 Test-and-set

Another concurrent object is test-and-set, for which we have presented a randomized logarithmic solution in Chapter 5. Our implementation is the fastest known against an adaptive adversary. Faster, sub-logarithmic solutions exist if the adversary is oblivious [6].

Test-and-set is computationally weaker than renaming, thus our time complexity lower bounds do not generalize to test-and-set. It does appear that logarithmic step complexity is the best achievable for this object using randomization against a strong adversary; on the other hand, proving this is an intriguing open question. On the other hand, recent results [6] show that there exist sub-logarithmic solutions if the adversary is oblivious. Exact bounds on the complexity of test-and-set against an oblivious adversary are also open.

# Bibliography

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, 1993.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[9] 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.

[10] 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.

[11] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: a fast array of wimpy nodes. *Communications of the ACM*, 54(7):101–109, 2011.

[12] 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.

[13] James Aspnes, Hagit Attiya, Keren Censor-Hillel, and Faith Ellen. Faster than optimal snapshots (for a while). Accepted to PODC 2012, February 2012.

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

[15] James Aspnes and Orli Waarts. Randomized consensus in expected $o(nlog^2 n)$ operations per processor. *SIAM J. Comput.*, 25(5):1024–1044, 1996.

[16] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. In *Proc. 9th Annual ACM symposium on Principles of Distributed Computing (PODC)*, pages 363–375. ACM, 1990.

[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 Keren Censor. Tight bounds for asynchronous randomized consensus. *Journal of the ACM*, 55(5):1–26, 2008.

[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, Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM*, 41(1):122–152, January 1994.

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

[22] 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.

[23] Hagit Attiya, Danny Hendler, and Philipp Woelfel. Tight RMR lower bounds for mutual exclusion and other problems. In *Proc. 40th Annual ACM Symposium on Theory of Computing (STOC)*, pages 217–226. ACM, 2008.

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

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

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

[27] 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.

[28] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proc. 2nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 27–30, August 1983.

[29] 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.

[30] Ken Birman. Replication and fault-tolerance in the ISIS system. In *Proc. 10th ACM Symposium on Operating Systems Principles (SOSP)*, pages 79–86, Orcas Island, WA USA, 1985.

[31] 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.

[32] 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.

[33] 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.

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

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

[36] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[37] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

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

[39] 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.

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

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

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

[43] Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[44] 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.

[45] Panagiota Fatourou and Nikolaos D. Kallimanis. A highly-efficient wait-free universal construction. In *Proc. 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 325–334. ACM, 2011.

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

[47] Faith Fich and Eric Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16(2-3):121–163, September 2003.

[48] 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.

[49] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

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

[51] Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *Proc. 43rd ACM Symposium on Theory of Computing (STOC)*, pages 373–382, 2011.

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

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

[54] Jae heon Yang and James H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9:9–1, 1994.

[55] Maurice Herlihy. Randomized wait-free concurrent objects (extended abstract). In *Proc. 10th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 11–21, New York, NY, USA, 1991. ACM.

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

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

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

[59] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[60] Prasad Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In *Proc. 17th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 201–210. ACM, 1998.

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

[62] Richard Karp, Christian Schindelhauer, Scott Shenker, and Berthold Vocking. Randomized rumor spreading. In *Proc. 41st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 565 –574, 2000.

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

[64] 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.

[65] 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.

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

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

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

[69] M.C. Loui and H.H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

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

[71] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press, New York, NY, USA, 2005.

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

[73] 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.

[74] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[75] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms.* Cambridge University Press, New York, NY, USA, 1995.

[76] Nathan Myhrvold. Moore's law corollary: Pixel power. *New York Times*, June 2006.

[77] Brian M. Oki and Barbara Liskov. Viewstamped replication: A general primary copy. In *Proc. 7th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 8–17, 1988.

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

[79] 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.

[80] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[81] Gary L. Peterson and Michael J. Fischer. Economical solutions for the critical section problem in a distributed system (extended abstract). In *Proc. 9th Annual ACM Symposium on Theory of Computing (STOC)*, pages 91–97. ACM, 1977.

[82] Serge Plotkin. Chapter 4: Sticky bits and universality of consensus. *Ph.D. Thesis, MIT*, 1998.

[83] Michel Raynal. *Algorithms for mutual exclusion.* MIT Press, Cambridge, MA, USA, 1986.

[84] Greg Semeraro, Grigorios Magklis, Rajeev Balasubramonian, David H. Albonesi, Sandhaya Dwarkadas, and Michael L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *Proc. 8th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 29 – 40, feb. 2002.

[85] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[86] Gadi Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

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

## Dan Alistarh

**Distributed Programming Laboratory (LPD),**
**School of Computer and Communication Sciences,**
**Swiss Federal Institute of Technology (EPFL)**
**d.alistarh@gmail.com**
**Web: http://people.epfl.ch/dan.alistarh**

### Education

| | |
|---|---|
| **09/2007 – present** | **Distributed Programming Laboratory (LPD), EPFL, Switzerland**<br>PhD student in Computer Science under the supervision of Prof. Rachid Guerraoui. |
| **09/2004 – 06/2007** | **Jacobs University, Bremen, Germany**<br>Double B.Sc. degree in Computer Science and Mathematics, on a merit-based scholarship from Jacobs University. |
| **08/2006 – 12/2006** | **Carnegie Mellon University, Pittsburgh, Pennsylvania, USA**<br>Semester abroad as an exchange student in the School of Computer Science. |

### Work and Project Experience

| | |
|---|---|
| **09/2007 - present** | **Research Assistant, Distributed Programming Laboratory (LPD), EPFL, Switzerland**<br>Research on the design, analysis and implementation of fast and reliable algorithms for distributed systems. The work resulted in new protocols and frameworks for secure information dissemination and efficient load balancing in message-passing systems (computer networks and mobile wireless networks), and optimal coordination and sharing mechanisms for concurrent shared-memory processors. |
| **01/2010 - 03/2010** | **Internship, T-Mobile Research Labs, Berlin, Germany**<br>Research project analyzing routing attacks and countermeasures in the context of distributed services deployed over the Internet. The project resulted in a set of efficient countermeasures, which were implemented in Java and tested on trace data. |
| **09/2006 - 12/2006** | **Systems Developer, Cylab Research Laboratory, Carnegie Mellon University, USA**<br>The aim of the project was to develop a 3-dimensional pointing device based on accelerometers. I was in charge of hardware design and of the software implementation using C#/.NET. The project resulted in a working prototype. |

### Awards

| | |
|---|---|
| **2007-2011** | Elected by the student body as the student representative in the EPFL Doctoral School.<br>The Best Paper Award at the International Conference on Distributed Computing and Networking (ICDCN), 2011. |
| **2004-2007** | Member of the Jacobs University President's List for high academic achievement.<br>Third prize at the International Mathematics Competition for University Students, 2005. |