Asynchronous Exclusive Selection *

Bogdan S. Chlebus † Dariusz R. Kowalski ‡

Abstract

We consider the task of assigning unique integers to a group of processes in an asynchronous distributed system of a total of n processes prone to crashes that communicate through shared read-write registers. In the Renaming problem, an arbitrary group of $k \leq n$ processes that hold the original names from a range $[N] = \{1, \ldots, N\}$, contend to acquire unique integers in a smaller range $[M]$ as new names using some r auxiliary shared registers. We develop a wait-free (k, N)-renaming solution, where both k and N are known, operating in $\mathcal{O}(\log k(\log N + \log k \log^* \frac{N}{k}))$ local steps, for $M = \mathcal{O}(k)$, and with $r = \mathcal{O}(k \log \frac{N}{k})$ auxiliary registers. We give a wait-free N-renaming algorithm, where N is known, operating in $\mathcal{O}(\log^2 k\,(\log N + \log k \log^* N))$ local steps, with $M = \mathcal{O}(k)$ and with $r = \mathcal{O}(n \log \frac{N}{n})$ registers. We develop a wait-free k-renaming algorithm, where k known, operating in $\mathcal{O}(k)$ time, for $M = 2k-1$ and with $r = \mathcal{O}(k^2)$ registers. We give an adaptive wait-free solution of Renaming, where neither k nor N is known, having $M = 8k - \lg k - 1$ as a bound on the range of new names, which operates in $\mathcal{O}(k)$ local steps and uses $r = \mathcal{O}(n^2)$ registers. As a lower bound, we show that a wait-free solution to Renaming requires $1 + \min\{k-2, \lfloor \log_{2r} \frac{N}{M+k-1} \rfloor\}$ steps in the worst case. We apply renaming algorithms to obtain solutions to Store&Collect problem, which is about a group of $k \leq n$ processes with the original names in a range $[N]$ proposing individual values (operation `Store`) and returning a view of all proposed values (operation `Collect`), while using some r auxiliary shared read-write registers. We show that if a known N is polynomial in n, then storing can be performed in $\mathcal{O}(\log^3 n \log^* n)$ local steps and collecting in $\mathcal{O}(k)$ local steps with $\mathcal{O}(n \log n)$ shared read-write registers. We consider a problem Mining-Names, in which processes may repeatedly request positive integers as new names subject to the constraints that no integer can be assigned to different processes and the number of integers never acquired as names is finite in an infinite execution. We give two solutions to Mining-Names in a distributed system in which there are infinitely many shared read-write registers available. A non-blocking solution leaves at most $2n - 2$ nonnegative integers never assigned as names, and a wait-free algorithm leaves at most $(n + 2)(n - 1)$ nonnegative integers never assigned as names.

Keywords: asynchrony, process crash, read-write shared register, renaming, store and collect, non-blocking algorithm, wait-free algorithm, deterministic algorithm, lower bound, graph expansion.

*A preliminary version of this paper appeared as [33].

†School of Computer and Cyber Sciences, Augusta University, Augusta, Georgia, USA. Work supported by the National Science Foundation Grant No. 1016847.

‡School of Computer and Cyber Sciences, Augusta University, Augusta, Georgia, USA. Work supported by the National Science Foundation Grant No. 2131538 and the Polish National Science Center NCN grant UMO-2017/25/B/ST6/02553.

1 Introduction

We consider asynchronous distributed systems consisting of some n processes that are prone to crashes and use read-write registers for inter-process communication. The studied problems concern assigning positive integers to the processes in an *exclusive* fashion, which means that no integer is assigned to two distinct processes. We seek wait-free algorithms, and sometimes consider non-blocking ones.

When an integer i is assigned to a process p exclusively, then we say that i is p's *new name*. In the Renaming problem, some $k \leq n$ processes, each having an original name from a large range $[N] = \{1, \ldots, N\}$, contend to acquire unique integers in a smaller range $[M]$ as new names, using some r shared registers. An algorithm can have some of the parameters k and N as a part of code. We indicate which parameters are known as a part of code by attaching the relevant parameters to problems' names and solutions. For example, an algorithm solving (k, N)-renaming has both k and N as a part of code, while M and r and the time complexity are characteristics of the algorithm, given as functions of k and N and possibly also of n. Similarly, an algorithm solving k-renaming works for any range $[N]$ of the original names and for up to k contending processes. Finally, an N-renaming algorithm handles the original names in the range $[N]$ while the contention k is arbitrary, except for the restriction $k \leq n$. An adaptive renaming algorithm works for any contention $k \leq n$ and range $[N]$ of the original names, which are not parts of code, while the range of new names M and the time performance are functions of k, and the number of registers r is a function of n, as this is the maximum value of k when $k \leq n$.

We restrict our attention to *one-time* renaming problems in which processes that will contend to acquire new names are designated at the start of an execution, and no new name is ever released and reused. Time performance is measure by the number of *local steps*, which is a maximum number of steps a process takes before halting in a final state.

In the problem Store&Collect, some k processes perform two operations `Store` and `Collect`. The `Store` operation by a process p proposes a value, and `Collect` results in returning a *view*, which is a collection of pairs (p, v) where p is the original name of a process that proposed a value v before the return of `Collect`, but not a stale one replaced before the invocation of `Collect`. The semantics of this problem under asynchrony is well determined by referring to a collection of read-write registers, one register assigned to each process. In order to propose a value, a process writes its original name and that value in its register. In order to collect, a process reads once each register storing a pair consisting of a value proposed by a process and its name, in arbitrary order of registers, and includes each such a read pair in the view.

The problem Mining-Names is about a scenario in which processes repeatedly request new positive integers as names in an infinite execution. There are two constraints on algorithms for the problem. One is that each process keeps an exclusive ownership of each acquired new name, to possibly build an infinite collection of new names in an infinite execution. The other is to leave only finitely many positive integers never assigned as new names in an infinite execution. For a fixed integer i, no wait-free solution to Mining-Names can guarantee that i is eventually assigned as a new name. It follows that some integers may never be used as new names when an algorithm works to assign as many positive integers as new names as possible. We want to minimize the number of positive integers never assigned as names in an execution, so this number is proposed as a measure of quality of a solution for Mining-Names. The model of a distributed system we use to develop

solutions for Mining-Names assumes finitely many processes but infinitely many shared read-write registers.

As a preparation to developing Mining-Names solutions, we show how to implement a repository of infinitely many values using infinitely many read-write registers. The values are generated in a dynamic fashion. A value is considered as *deposited* in a register when the value is stored in the register and will never be overwritten. In this problem, we strive to minimize the number of available registers that never become used to store deposited values. The problem is closely related to mining names, as new names can be used to identify registers to make deposits.

Contributions of this paper. The renaming algorithms that we develop are designed to have processes traverse paths in graphs in which vertices represent names. During such concurrent traversals, processes compete to acquire the names of the visited vertices. We consider graphs with suitable expansion properties as means to makes algorithms efficient. The approaches to renaming known in the literature, that have graphs built into algorithms in a similar manner, use graphs with simple regular topologies, like constant-degree grids.

We develop a wait-free (k, N)-renaming algorithm operating in $\mathcal{O}(\log k(\log N + \log k \log^* \frac{N}{k}))$ local steps, with a range of new names $M = \mathcal{O}(k)$ and $r = \mathcal{O}(k \log \frac{N}{k})$ auxiliary registers. This is a first known deterministic algorithm with step complexity polylogarithmic in k and a range of new names M linear in k, for N that is polynomial in k.

We show that $1 + \min\{k - 2, \lfloor \log_{2r} \frac{N}{M+k-1} \rfloor\}$ local steps are required in the worst case by any wait-free (k, N)-renaming algorithm to assign names from a range $[M]$ when using r registers. This is a first known lower bound on the local-step time performance of Renaming that comprehensively involves all the four parameters k, N, M, and r. In particular, if N is unknown, and hence could be arbitrarily large, while M is suitably bounded as a function of k, then $k - 1$ is a lower bound; this resembles the lower bounds given by Jayanti et al. [41]. An $\Omega(k)$ lower bound on time, valid under additional assumptions, was proven by Alistarh et al. [10] by a different argument.

We develop a wait-free N-renaming algorithm with $\mathcal{O}(\log^2 k\,(\log N + \log k \log^* N))$ local step complexity, for unknown contention k, with the range of new names $M = \mathcal{O}(k)$ and the number of registers $r = \mathcal{O}(n \log \frac{N}{n})$. If a known N is poly-logarithmic in n, then this renaming algorithm runs in $\mathcal{O}(\log^3 n \log^* n)$ local steps and uses $\mathcal{O}(n \log n)$ auxiliary registers.

We develop a wait-free k-renaming algorithm operating in $\mathcal{O}(k)$ local steps, with a bound on new names $M = 2k - 1$ and with $r = \mathcal{O}(k^2)$ auxiliary registers. The time complexity of this algorithm is asymptotically optimal, which follows from the lower bound we show and the property that the algorithm works for arbitrary N. This is the first algorithm known that has simultaneously two properties: the local step complexity is $\mathcal{O}(k)$ and also the range of new names is $M = \mathcal{O}(k)$. Among the previously known algorithms that run in time $\mathcal{O}(k)$, the value $M = \frac{k(k+1)}{2}$ was smallest known; it is achieved by an algorithm of Moir and Anderson [45]. The fastest algorithm known before, among those having $M = \mathcal{O}(k)$, operates in $\mathcal{O}(k \log k)$ time, it was given by Attiya and Fouren [18]. The value $M = 2k - 1$ is known to be the best possible size of a range of names for infinitely many values of k, as showed by Herlihy and Shavit [40]. The fastest algorithm known prior to this work with $M = 2k - 1$ as a bound on the range of new names runs in time $\mathcal{O}(k^2)$, it was given by Afek and Merritt [5].

We give a fully adaptive renaming algorithm, with neither k nor N known, having a bound on the range of new names M as small as $8k - \lg k - 1$. The algorithm operates in $\mathcal{O}(k)$ local steps and

uses $\mathcal{O}(n^2)$ auxiliary registers. This is an improvement with respect to time performance over the previously known algorithms. The algorithm of Moir and Anderson [45] works in time $\mathcal{O}(k)$ with a range of new names $M = \mathcal{O}(k^2)$ and using $\mathcal{O}(n^2)$ registers. The algorithm of Attiya and Fouren [18] operates in $\mathcal{O}(k \log k)$ time with a range $M = \mathcal{O}(k)$. The algorithm of Afek and Merritt [5] runs in time $\mathcal{O}(k^2)$ with a range of new names $M = 2k - 1$.

We apply renaming algorithms to obtain solutions to Store&Collect of the following performing characteristics. When both parameters k and N are known, then Store&Collect can be implemented such that the first storing operation by a process takes $\mathcal{O}(\log k(\log N + \log k \log^* \frac{N}{k}))$ steps and collecting $\mathcal{O}(k)$ steps, while using $\mathcal{O}(k \log \frac{N}{k})$ auxiliary registers. If N is known but k is not, then Store&Collect can be implemented such that the first instance of storing takes $\mathcal{O}(\log^2 k(\log N + \log k \log^* N))$ local steps and collecting $\mathcal{O}(k)$ steps, with $\mathcal{O}(n \log \frac{N}{n})$ auxiliary registers. If the number of participating processes k is known but the range of original names N is not, then Store&Collect can be implemented such that the first instance of storing takes $\mathcal{O}(k)$ local steps and collecting $\mathcal{O}(k)$ steps, with $\mathcal{O}(k^2)$ auxiliary registers. If k and N are both unknown, which is the adaptive case, then Store&Collect can be implemented such that storing takes $\mathcal{O}(k)$ steps and collecting takes $\mathcal{O}(k)$ steps, with $\mathcal{O}(n^2)$ auxiliary registers. Afek and De Levie [4] gave an adaptive solution to Store&Collect achieving storing in $\mathcal{O}(k)$ local steps and collecting in $\mathcal{O}(k)$ local steps, for $r = \mathcal{O}(n^2)$.

We consider the problem called Mining-Names, which is about processes working to claim nonnegative integers as names in a mutually exclusive manner. We show that Mining-Names is solvable in a non-blocking way such that at most $2n-2$ integers are never assigned as names, which is asymptotically best possible, and in a wait-free manner so that at most $(n+2)(n-1)$ values are never assigned as names. The problem Mining-Names has not been considered before in the literature, by the knowledge of the authors of the paper. Name mining is related to "depositing" infinitely many values in read-write registers, where depositing means storing a value in a register such that it is never overwritten. We give a non-blocking implementation of depositing in which at most $2n - 2$ dedicated read-write registers are never used for depositing, and a wait-free implementation with the property that at most $(n + 2)(n - 1)$ dedicated deposit registers are never used for depositing.

Related work. Now we describe the context of this work by reviewing research related to renaming. We restrict our attention to asynchronous systems with shared memory consisting of read-write registers only; for a comprehensive survey of renaming see Alistarh [9].

The problem of renaming was introduced by Attiya et al. [15] in the model of asynchronous message-passing. They showed that n processes may assign themselves new names from the range $[n + f]$, where $f < n$ is an upper bound on the number of crashes. This established renaming as a non-trivial algorithmic problem with a wait-free solution for environments in which Consensus cannot be solved; see [24, 39, 42]. The range of new names $[M = n + f]$, with up to f crashes, was shown to be the smallest possible for renaming to be solvable by Herlihy and Shavit [40]. Next we consider a scenario win which $k \leq n$ contending processes with original names in a large range $[N]$ want to obtain new names in a small range $[M]$. Borowsky and Gafni [25] gave a wait-free algorithm solving this problem in time $\mathcal{O}(k^2 N)$ for $M = 2k - 1$. Moir and Anderson [45] gave a solution with time complexity $\mathcal{O}(k)$, for new names of magnitude $M = \frac{k(k+1)}{2}$ and using $r = \mathcal{O}(k^2)$ registers. Attiya and Fouren [18] gave an algorithm working in time $\mathcal{O}(k \log k)$ for $M = 6k - 1$, and another of time complexity $\mathcal{O}(N)$ for $M = 2k - 1$. Afek and Merritt [5] developed an algorithm working in time $\mathcal{O}(k^2)$ for $M = 2k - 1$.

A renaming solution is *long-lived* when processes may invoke the operations to request a name and to release the current name repeatedly, as long as exclusiveness of each name holds within the interval from acquiring to releasing. It is assumed that at most k processes contend for names concurrently. The following is a selection of published long-lived renaming algorithms. Burns and Peterson [28] gave a solution of time complexity $\mathcal{O}(Nk^2)$, for $M = 2k-1$ and $r = \mathcal{O}(N^2)$. Moir and Anderson [45] improved the time to $\mathcal{O}(Nk)$, for $M = \frac{k(k+1)}{2}$ and $r = \mathcal{O}(Nk^2)$. Further improvements were due to Buhrman et al. [27], who achieved $\mathcal{O}(k^3)$ time, for $M = \frac{k(k+1)}{2}$ and the number of registers $r = \mathcal{O}(k^4 \min\{3^k, N\})$, and to Moir and Garay [46] whose algorithm achieved $\mathcal{O}(k^2)$ time complexity, for $M = \frac{k(k+1)}{2}$ and $r = \mathcal{O}(k^3)$ registers, and who also gave another solution with $\mathcal{O}(k^4)$ time, for $M = 2k-1$ and $r = \mathcal{O}(k^4)$.

For other work on renaming, see the papers by Afek et al. [2, 3, 7], Brodsky et al. [26], and Eberly et al. [36]. Randomized renaming algorithms were considered by Alistarh et al. [11, 12, 13]. Lower bounds on renaming were given by Alistarh et al. [10], Attiya et al. [16, 22, 23], Burns and Peterson [28], Castañeda et al. [30], Castañeda and Rajsbaum [31, 32], and Helmi et al. [38].

Previous work on the problem Store&Collect includes papers by Afek et al. [6], Attiya et al. [17, 19, 20, 21], Chlebus et al. [34] and Saks et al. [47]. Previous work on models with infinite arrivals of processes and infinitely many shared registers includes [8, 14, 35, 37, 43, 44].

2 Technical Preliminaries

Algorithms are executed in an asynchronous system with n processes prone to crashes and a set of read-write registers. Each process p is identified by its *original name* $\texttt{name}_p$, which is a unique number in some range of names $[N] = \{1, \ldots, N\}$.

If a parameter of a distributed system can be used in a code of algorithm then this parameter is *known*. In particular, each process p knows its original name, which is referred to by a specialized variable in codes of algorithms, say `name`, with process p substituting $\texttt{name}_p$ as its private value. We assume throughout that the number n is known.

The following is a standard terminology regarding delays of enabled operations; see [24, 39, 42] for expositions of these and related concepts. When, for any configuration in an execution, some process will eventually complete an invoked operation, then the executed algorithm is *non-blocking.* When, for any configuration in an execution, each process will eventually complete an invoked operation, even when all the remaining processes have crashed, then the algorithm is *wait-free.*

Competing for registers. We introduce a procedure used by processes to compete for a shared register. The procedure has two two properties. First is that a lack of contention guarantees wining. This means that if there is exactly one process p working to win a register R, then p eventually wins R. The second property is that a win provides exclusivity. This means that once some contender wins a register R, then no other contender will ever win R. This specification does not require a register to be won by a process when there are multiple contenders but also does not preclude this.

To implement a competition for a register R, we use an auxiliary dedicated shared register H_R initialized to `null`. This register H_R is used as a placeholder to store a reservation for R. A pseudocode of a procedure for p is given in Figure 1.

procedure COMPETE-FOR-REGISTER (R)

1. read: $\texttt{contention}_p \leftarrow R$
2. `if` $\texttt{contention}_p = \texttt{null}$ `then` write $R \leftarrow \texttt{name}_p$ `else exit`
3. read: $\texttt{contention}_p \leftarrow H_R$
4. `if` $\texttt{contention}_p = \texttt{null}$ `then` write $H_R \leftarrow \texttt{name}_p$ `else exit`
5. read: $\texttt{contention}_p \leftarrow R$
6. `if` $\texttt{contention}_p = \texttt{name}_p$ `then return win else exit`

Figure 1: Pseudocode for a process p with name $\texttt{name}_p$ to win a shared register R. It uses a private variable $\texttt{contention}_p$ and a shared register H_R associated with R. Both registers R and H_R are initialized to `null`.

Lemma 1 *Procedure* COMPETE-FOR-REGISTER() *is an implementation of competition for a register.*

Proof: To show correctness, consider two cases corresponding to the specification during a competition for a register R. If a process p acts as the only contender to win R, then p eventually writes the value $\texttt{name}_p$ to both registers H_R and R, and the final read from R makes process p a winner. Now suppose some process p wins R in the presence of a contender process q. If the first read of R by q does not return `null` then q exits immediately and no longer contends to win R. Otherwise, process q reads `null` from R as its first action, which means that process q managed to read from register R before process p wrote to R. Still process p wins R, so p managed to write $\texttt{name}_p$ to R and then the same value to H_R and then check that $\texttt{name}_p$ is still in register R, as all this is required to win R. When process p confirms by the second read of R that $\texttt{name}_p$ is still there then H_R already stored $\texttt{name}_p$. So process q could overwrite the value $\texttt{name}_p$ at register R only after process p read it for the second time, which means after register R has the value $\texttt{name}_p$ stored in it already. When process q reads register H_R then this occurs after process p wrote $\texttt{name}_p$ to it, so the read returns $\texttt{name}_p$, which is different from `null`. This makes process q exit without invoking a write to H_R. □

Graphs. Let $G = (V, W, E)$ be a simple bipartite graph. This notation means that the vertices are partitioned into the set of *inputs* V and the set of *outputs* W, E is the set of edges, and each edge has one endpoint in V and the other in W. We say that graph G has *input-degree* Δ if each vertex in V is connected to exactly Δ neighbors in W. A graph G is said to be an (L, Δ, ε)-*lossless expander*, for a natural number L, if Δ is the input-degree of G and each subset X of V of size $|X| \leq L$ has at least $(1 - \varepsilon)\Delta \cdot |X|$ neighbors in W. A vertex $v \in W$ is a *unique neighbor of set* $S \subseteq V$ if vertex v is adjacent to exactly one vertex in S.

Lemma 2 ([29]) *Let G be a (L, Δ, ε)-lossless expander, for some parameters L and $\varepsilon < \frac{1}{2}$. Then, for each subset $X \subseteq V$ of size $|X| \leq L$, at least the fraction $(1 - 2\varepsilon)\Delta$ of vertices among the neighbors of X are unique neighbors.*

Proof: Fix an ordering of the sets of vertices W. For $v \in V$, let (v, i) denote the ith neighbor of v in W. There are $\Delta|X|$ pairs of the form (v, i), for $v \in X$ and $1 \leq i \leq \Delta$. By the definition of lossless expanders, all these pairs determine at least $(1 - \varepsilon)\Delta|X|$ neighbors of X. It follows that at most

$\varepsilon\Delta|X|$ such pairs denote vertices repeated at least twice. Each such a repetition $(v_1, i) = (v_2, j)$ eliminates two possible unique neighbors of X per one real neighbor $(v_1, i) = (v_2, j)$, in the total number $\Delta|X|$ of pairs of the form (v, i), for $v \in X$ and $1 \leq i \leq \Delta$. □

Lemma 3 *If a bipartite graph $G = (V, W, E)$ is an (L, Δ, ε)-lossless expander, for some numbers L and $\varepsilon < \frac{1}{2}$, then, for each set $X \subseteq V$ of size $|X| \leq L$, there is a partial matching in G, between some vertices in X and unique neighbors of X, that has at least $(1 - 2\varepsilon)|X|$ edges.*

Proof: By Lemma 2, a fraction of at least $(1 - 2\varepsilon)\Delta$ vertices among the neighbors of X are unique neighbors. We can match these inputs to their unique neighbors. □

Let $\lg z$ denote the logarithm of z to the base 2, and e be the base of natural logarithms. We will use the existence of lossless expanders with the following properties:

Lemma 4 *Let V and W be two finite disjoint sets and L a natural number such that $1 \leq L \leq \frac{|V|}{2}$. If $|W| = 12e^4 L \lg \frac{|V|}{L}$ then there exists a bipartite graph $G = (V, W, E)$ of input-degree Δ such that $4 \leq \Delta \leq 4 \lg |V|$ and G is a $(L, \Delta, \frac{1}{4})$-lossless expander.*

Proof: We show that a set of edges between the vertices in V and W selected randomly subject to degree constraints meets the requirements with a positive probability. More precisely, for each vertex $v \in V$, we select Δ neighbors in W uniformly at random. Next, we demonstrate that the resulting graph is a $(L, \Delta, \frac{1}{4})$-lossless-expander with a probability greater than 0, where a specific value of Δ will be determined later.

Let $X \subseteq V$ be a subset of $x = |X| \leq L$ elements, and $Y \subseteq W$ be a subset of $\frac{3}{4}x\Delta$ elements. The probability that all the neighbors of X are in the set Y is at most

$$\left(\frac{\binom{|Y|}{\Delta}}{\binom{|W|}{\Delta}} \right)^x \leq \left(\frac{|Y|e}{|W|} \right)^{x\Delta} \leq \left(\frac{\frac{3e}{4} \cdot x\Delta}{|W|} \right)^{x\Delta},$$

where we used the bounds $(\frac{n}{n})^k \leq \binom{n}{k} \leq (\frac{en}{k})^k$. The number of different subsets $X \subseteq V$ of size x is at most

$$\binom{|V|}{x} \leq \left(\frac{|V|e}{x} \right)^x .$$

The number of different subsets $Y \subseteq W$ of size $\frac{3}{4}x\Delta$ is at most

$$\binom{|W|}{\frac{3}{4}x\Delta} \leq \left(\frac{|W|e}{\frac{3}{4}x\Delta} \right)^{3x\Delta/4} .$$

Therefore, the probability that there exists a set $X \subseteq V$ with at most $\frac{3}{4}x\Delta$ neighbors, for a given size x, is at most

$$\left(\frac{|V|e}{x} \right)^x \cdot \left(\frac{|W|e}{\frac{3}{4}x\Delta} \right)^{3x\Delta/4} \cdot \left(\frac{\frac{3e}{4}x\Delta}{|W|} \right)^{x\Delta} \leq \left(\frac{|V|e}{x} \right)^x \cdot \left(\frac{\frac{3e^3}{4}x\Delta}{|W|} \right)^{x\Delta/4} . \tag{1}$$

We assumed $|W| = 12e^4 L \lg \frac{|V|}{L}$. Now, define $\Delta = 4 \lg \frac{|V|}{L}$. Observe that $\Delta \geq 4$, because $|V| \geq 2L$. The assumption on W and the specification fo Δ produce cancellations that give:

$$\frac{3e^4}{4} \cdot \frac{x\Delta}{W} = \frac{3e^4}{4} \cdot \frac{x4 \lg \frac{|V|}{L}}{12e^4 L \lg \frac{|V|}{L}} = \frac{x}{4L} .$$

To estimate (1) further, observe that the quantity raised to the power x in (1) equals:

$$\frac{|V|}{L} \cdot \frac{L}{x} \cdot \left(\frac{3e^4 x\Delta}{4|W|}\right)^{\Delta/4} = \frac{|V|}{L} \cdot \frac{L}{x} \cdot \left(\frac{1}{4} \cdot \frac{x}{L}\right)^{\lg \frac{|V|}{L}} . \tag{2}$$

Let us introduce the notation $\alpha = \frac{|V|}{L}$ and $\beta = \frac{L}{x}$. By the assumptions, we have $\alpha \geq 2$ and $\beta \geq 1$. The right-hand side of (2) can be represented as

$$\frac{\alpha\,\beta}{(4\beta)^{\lg \alpha}} = \frac{\beta}{\alpha\,\beta^{\lg \alpha}} .$$

This quantity is at most $\frac{1}{2}$, for $\alpha \geq 2$ and $\beta \geq 1$. It follows that the right-hand side of (1) is at most $\frac{1}{2^x}$.

We demonstrated that the probability that some subset $X \subseteq V$ of size $x = |X| \leq L$ has at most $\frac{3}{4}x\Delta$ neighbors in W is at most $\frac{1}{2^x}$. The probability that some subset $X \subseteq V$ of size $x = |X| \leq L$ has at most $\frac{3}{4}|X|\Delta$ neighbors in W is at most $\sum_{x=1}^{L} \frac{1}{2^x} < 1$. By the probabilistic method, there exists a bipartite graph $G = (V, W, E)$ with input-degree $\Delta = 4 \lg \frac{|V|}{L}$ in which every subset $X \subseteq V$ of size $|X| \leq L$ has more than $\frac{3}{4}|X|\Delta$ neighbors in W, where the number L satisfies $L \leq \frac{|V|}{2}$. □

3 Bounded Selection

We consider the problem of assigning new names to a group of participating processes, from among the total of n, known as *renaming*. Each among n processes holds an *original* name from some range $[N] = \{1, \ldots, N\}$, which is the value of its variable $\texttt{name}_p$. The number of participating processes is denoted as k, where $k \leq n$. These participating processes contend to acquire unique integers in a range $[M]$ as new names using some r auxiliary shared registers, where $M < N$. The goal is to minimize a number of parameters: the running time and a range $[M]$ of new names, but also the number of auxiliary registers r. Running time means local steps, which is a maximum among the processes of the number of time-steps counted by each process until halting.

We assume that n is known, but whether k and N are known depends on a precise specification of the renaming problem. This leads to four variants of the problem of renaming, where either (1) both k and N are known, or (2) only k is known, or (3 only N is known, and finally (4) with none of k and N known. The case of both k and N unknown is most challenging. Algorithms for renaming that do not refer to either k nor N in their codes, and so work for arbitrary unknown values of k and N, are called *adaptive*. We apply the convention to add known parameters among k and N to names of algorithms, so that if a parameter is missing in the name then this means the parameter is unknown.

Our first goal is to develop a renaming algorithm with both k and N known. We begin by introducing an auxiliary problem called Majority-Renaming, which is about assigning new names to at least half of some k contending processes. An algorithm is *(k, N)-majority renaming with a bound M on new names* if at least half of any k contending processes with original names in $[N]$ acquire unique names in $[M]$, while the numbers k and N can be part of code of the algorithm. We find a solution for Majority-Renaming based on lossless expanders with good unique-neighbors properties. This becomes a stepping stone to develop solutions for Renaming itself. Employing a

renaming algorithm that relies on some known information, we then argue how to obtain an adaptive solution. Finally, we discuss how to use the obtained renaming algorithms to solve Store&Collect.

We begin by presenting an algorithm called MAJORITY(ℓ, N), where N is a natural number and $\ell \leq \frac{N}{2}$, which is (ℓ, N)-majority renaming. The number $M = 12e^4 \ell \lg \frac{N}{\ell}$ serves as a bound on the range of new names. We consider a bipartite graph $G = (V, W, E)$, where $V = [N]$, $W = [M]$, and each input-degree is Δ. The topology of G is such that G satisfies the properties given in Lemma 4. The graph G is part of code of the algorithm. The set V of inputs corresponds to all original names of processes. Each output vertex in the set W represents a possible new name.

The edges of G determine which registers will be competed for by the processes, using procedure COMPETE-FOR-REGISTER given in Figure 1. To facilitate this, there are two unique shared registers associated with each output vertex.

Competition to win registers in an execution of algorithm MAJORITY(ℓ, N) proceeds as follows. A process p with a name in $V = [N]$ uses a vertex in V labeled $\texttt{name}_p$. It begins by attempting to win a register corresponding to the first neighbor of $\texttt{name}_p$ in W. If p fails then it attempts to win the register of its second neighbor in W. This continues until p either wins a register or exhausts all the neighbors. As soon as p wins a register in W then p adopts the number of the won register as its new name and exits. If p fails all Δ competitions, then p halts without acquiring a new name.

Lemma 5 *Algorithm* MAJORITY(ℓ, N)*, for a natural number* N *and* $\ell \leq \frac{N}{2}$*, is* (ℓ, N)*-majority-renaming, where* $M = 12e^4 \ell \lg \frac{N}{\ell}$ *is a bound on new names. The algorithm operates in* $\mathcal{O}(\log N)$ *local steps and uses* $24e^4 \ell \lg \frac{N}{\ell}$ *auxiliary registers.*

Proof: The graph G is a $(\ell, \Delta, \frac{1}{4})$-lossless-expander, as stated in Lemma 4. A majority of contending processes have unique neighbors not shared with other active processes, by Lemma 3 applied to graph G. If an active process has a unique neighbor then it eventually wins some register representing its neighbor, by Lemma 1. Hence a majority of active processes get unique names. The worst-case running time is proportional to the degree Δ of graph G, which is $\mathcal{O}(\log N)$. The number of used registers is $2M$, as we use two unique shared registers per one vertex in W. □

Renaming when both k and N are known. Next, we consider an algorithm PLAIN-RENAME(k, N), which is (k, N)-renaming with $M = 24e^4 k \lg \frac{N}{k}$ as a bound on new names. A process $p \in [N]$ proceeds through consecutive stages, up to $1 + \lg k$ stages maximum, until it gets a unique name. In a stage i, where $0 \leq i \leq \lg k$, process p executes MAJORITY$(\frac{k}{2^i}, N)$ on the set of pairs of registers M_i, where $|M_i| = 12e^4 \frac{k}{2^i} \lg \frac{N 2^i}{k}$. We assume that the sets M_i are mutually disjoint. A union of these sets $\cup_{i=1}^{1+\lg k} M_i$ constitutes a collection of new names.

Lemma 6 *Algorithm* PLAIN-RENAME(k, N) *is* (k, N)*-renaming with* $M = 24e^4 k \lg \frac{2N}{k}$ *as a bound on new names. It operates in* $\mathcal{O}(\log k \log N)$ *local steps and uses* $48e^4 k \lg \frac{2N}{k}$ *auxiliary registers.*

Proof: Procedure MAJORITY$(\frac{k}{2^i}, N)$ is majority renaming, by Lemma 5. Each call of this procedure at least halves the number of contending processes that still need names. Calls of the procedure continue until there remain at most one process without a new name, which then eventually also gets a name. This takes $\mathcal{O}(\log k \cdot \log N)$ local steps in total, by Lemma 5. Number $1 + \lg k$ is a

bound on the number of stages. The size of a pool of new names can be estimated as follows:

$$M = \sum_{i=0}^{\lg k} 12e^4 \frac{k}{2^i} \lg \frac{2^i N}{k} = 12e^4 k\Big(\sum_{i=1}^{\lg k} \frac{i}{2^i} + \sum_{i=0}^{\lg k} \frac{1}{2^i} \lg \frac{N}{k}\Big) = 12e^4 k\big(2 + 2\lg \frac{N}{k}\big) = 24e^4 k \log \frac{2N}{k} \ .$$

The number of shared registers needed for this to work correctly is $2M$. □

Next, we consider an (k, N)-renaming algorithm called Compact-Rename(k, N). It is an improvement over Plain-Raname in that the range of new names is $\mathcal{O}(k)$.

An execution of Compact-Rename(k, N) is structured as a sequence of epochs. A process $p \in [N]$ participates in consecutive epochs, in each one getting a new name. At least one epoch is executed as long as $N \geq 2^{15}k$, otherwise the original names serve as new names without any change. The original names are used in the first epoch, and then the names acquired in epoch i are used as original names in epoch $i + 1$. This continues until the upper bound on the range of the new names, determined by the properties of a current epoch, becomes less than $2^{15}k$. A process acquires its ultimate name during this last epoch.

In epoch j, process p executes Plain-Rename(k, N_j), where $N_1 = N$ and $N_{j+1} = 24e^4 k \lg \frac{2N_j}{k}$ are bounds on new names in calls of Plain-Rename(k, N_j), for $j \geq 1$. The executions of instantiations of algorithm Plain-Rename use sets of shared registers dedicated for each epoch, such that a shared register is used in only one epoch. Processes use names from the range $[N_j]$ in epoch j, with $N_1 = N$. The names assigned in epoch j are from the range $[N_{j+1}]$.

The algorithm Compact-Rename(k, N) terminates because the ranges of new names shrink with passing epochs: $N_{j+1} < N_j$, for $j > 0$. We evaluate the rate of shrinking next.

Lemma 7 *If $N \geq 2^{15}k$ then $\frac{N_2}{N_1} < \frac{2}{3}$, and as long as $N_{j-1} \geq 2^{15} \cdot k$ then $\frac{N_{j+1}}{N_j} < \frac{24}{25}$, for $j > 1$.*

Proof: The case of $j = 1$:

$$\frac{N_2}{N_1} = \frac{24\, e^4\, k \lg \frac{2N}{k}}{N} \leq \frac{24\, e^4\, k \lg \frac{2^{16}k}{k}}{2^{15}k} = \frac{24\, e^4\, 16}{2^{15}} < \frac{2}{3} \ .$$

The case of $j > 1$:

$$\frac{N_{j+1}}{N_j} = \frac{24e^4 k \lg \frac{2N_j}{k}}{24e^4 k \lg \frac{2N_{j-1}}{k}} = \frac{\lg(48e^4 \lg \frac{2N_{j-1}}{k})}{\lg \frac{2N_{j-1}}{k}} = \frac{\lg(48e^4)}{\lg \frac{2N_{j-1}}{k}} + \frac{\lg \lg \frac{2N_{j-1}}{k}}{\lg \frac{2N_{j-1}}{k}} < \frac{71}{100} + \frac{1}{4} = \frac{24}{25} \ ,$$

for $N_{j-1} \geq 2^{15} \cdot k$. □

Define a sequence $(a_j)_{j \geq 1}$: $a_j = \lg(\frac{2N_j}{k})$ for $j \geq 1$.

Lemma 8 *If $N_j \geq 2^{16}k$ then $a_{j+1} < \log_{\frac{3}{2}} a_j$.*

Proof: The sequence a_j satisfies the following recurrence, for $j \geq 1$:

$$a_{j+1} = \lg\Big(\frac{2N_{j+1}}{k}\Big) = \lg(48e^4 \lg\Big(\frac{2N_j}{k}\Big)) = \lg(48e^4 a_j) \ .$$

The inequality $\lg(48e^4 x) < \log_{\frac{3}{2}} x$ holds for $x \geq 17$, by inspection. Substituting $a_j = \lg(\frac{2N_j}{k})$ for x, we obtain that $a_{j+1} < \log_{\frac{3}{2}} a_j$ for $\lg(\frac{2N_j}{k}) \geq 17$, which holds for $N_j \geq 2^{16}k$. □

We use the iterated-logarithm function $\log^{(i)} n$, which denotes $\log n$ iterated i times. A recursive definition of this function reads $\log^{(0)} n = n$ and $\log^{(i+1)} n = \log\log^{(i)} n$. We also refer to $\log^* n = \min\{i : \log^{(i)} n \leq 1\}$, for $n > 1$.

Theorem 1 *Algorithm* Compact-Rename(k, N) *is (k, N)-renaming with $M = 2^{15}\, k$ as a bound on new names, assuming $N \geq 2^{15}\, k$. It operates in $\mathcal{O}(\log k(\log N + \log k \log^* \frac{N}{k}))$ local steps and uses $\mathcal{O}(k \log \frac{N}{k})$ auxiliary registers.*

Proof: We assume that $N \geq 2^{15}k$, as otherwise no epoch is executed. The ranges of names used through the epochs can be traced back to $N_1 = N$ as follows. If $N_j \geq 2^{16}k$ then

$$N_{j+1} = 24e^4 k \lg \frac{2N_j}{k} = 24e^4 k a_j < 24e^4 k \log_{\frac{3}{2}}^{(j-i)} a_i \ ,$$

as long as $N_{j-i} \geq 2^{16}k$, by Lemma 8. Combining this with Lemma 7, we obtain the bound

$$N_{j+1} = \mathcal{O}\Big(k \log^{(j)} \frac{N}{k}\Big) \ ,$$

for $N \geq 2^{15}k$. The final range of new names $[N_{j^*+1}]$ satisfies $N_{j^*+1} < 2^{15}k$. The number of epochs is $j^* = \mathcal{O}(\log^* \frac{N}{k})$.

The number of local steps can be estimated as follows:

$$\mathcal{O}\big(\sum_{j=1}^{j^*} \log k \log N_j\big) = \mathcal{O}\big(\log k \sum_{j=1}^{j^*} (\log k + \log^{(j)} N)\big) = \mathcal{O}(\log^2 k \cdot \log^* \frac{N}{k} + \log k \cdot \log N) \ ,$$

by Lemma 6 and the bound on the number of epochs $j^* = \mathcal{O}(\log^* \frac{N}{k})$.

The first epoch uses $48e^4 \lg \frac{2N}{k}$ registers. A number of registers used in subsequent epochs is given by Lema 6. These numbers keep decreasing, with a rate determined Lemmas 7 and 8. By combining this together we obtain that the number of needed registers is

$$\sum_{j=1}^{j^*+1} 48e^4 \lg \frac{2N_j}{k} = 48e^4 k \sum_{j=1}^{j^*+1} a_j \ .$$

The estimate on the rate of decreasing of a_j given in Lemma 8 applies for all but $\mathcal{O}(1)$ epochs. It follows that the number of needed registers is $\mathcal{O}(ka_1) = \mathcal{O}(k \log \frac{N}{k})$. □

The knowledge of k and range N that is polynonomial in n allows to obtain a renaming algorithm efficient with respect to the total number of processes n.

Corollary 1 *If k and N are known and $N = \mathcal{O}(n)$, then algorithms* Compact-Rename(k, N) *runs in $\mathcal{O}(\log^2 n)$ local steps and uses $\mathcal{O}(n)$ auxiliary registers. If k and N are known and N is polynomial in n, then algorithm* Compact-Rename(k, N) *runs in $\mathcal{O}(\log^2 n \log^* n)$ local steps and uses $\mathcal{O}(n \log n)$ auxiliary registers.*

Proof: Algorithm Compact-Rename(k, N) needs $\mathcal{O}(\log k(\log N + \log k \log^* \frac{N}{k}))$ local steps, by Theorem 1. This bound is $\mathcal{O}(\log^2 n)$ if $N = \mathcal{O}(n)$, and it is $\mathcal{O}(\log^2 n \log^* n)$ if N is polynomial in n. Algorithm Compact-Rename(k, N) uses $\mathcal{O}(k \log \frac{N}{k})$ auxiliary registers, by Theorem 1. If a known N is such that $N = \mathcal{O}(n)$ then use $\mathcal{O}(k \log \frac{n}{k}) = \mathcal{O}(n)$, which holds for $1 \leq k \leq n$, to obtain the bound $\mathcal{O}(n)$ on the number of registers. If a known N satisfies $N = \mathcal{O}(n^\alpha)$, for $\alpha > 1$, then the bound on the number of registers becomes $\mathcal{O}(k \log \frac{N}{k}) = \mathcal{O}(n \log n)$. □

Renaming when N is known while k is not. We present an algorithm RANGE-RENAME(N), which renames k contending processes assigning names of magnitude $\mathcal{O}(k)$. A bound N on the magnitude of original names is known but the number of participating processes k is unknown.

The algorithm is specified as follows. A process participates in consecutive executions of algorithms COMPACT-RENAME($2^i, N$), for consecutive integers $i = 1, 2, 3 \ldots$, until it obtains a new name. After the first execution is over, it starts a new execution only after the previous one has failed to assign a new name. These consecutive executions use disjoint sets of registers. An execution of COMPACT-RENAME($2^i, N$), for $i > 1$, uses a next interval of integers as a range of new names, following the intervals used by COMPACT-RENAME($2^j, N$), for $1 \leq j < i$. This means the size of interval used by COMPACT-RENAME($2^i, N$) is $2^{15}2^i$, as indicated by Theorem 1, but it is of the form $[\ell, \ell + 2^{15}2^i]$, for a suitable positive integer ℓ that is greater than 1, except for $i = 1$.

Theorem 2 *Algorithm* RANGE-RENAME(N) *is N-renaming. It assigns new names of magnitude $2^{16}k$, runs in $\mathcal{O}(\log^2 k(\log N + \log k \log^* N))$ local steps, and uses $\mathcal{O}(n \log \frac{N}{n})$ auxiliary registers.*

Proof: At most k processes participate in each execution of COMPACT-RENAME($2^i, N$). We have that $i \leq \lceil \lg k \rceil$, because as soon as $k \leq 2^i$ then each process acquires a new name. The size of the range of new names is estimated by Theorem 1 to be at most

$$2^{15} \sum_{j=1}^{\lceil \lg k \rceil} 2^j \leq 2^{16}k \ .$$

The number of steps through executing COMPACT-RENAME($2^{\lceil \lg k \rceil}, N$) follows from Theorem 1:

$$\mathcal{O}\Big(\sum_{j=1}^{\lceil \lg k \rceil} \Big(j \log N + j^2 \log^* \frac{N}{2^j} \Big) \Big) = \mathcal{O}(\log^2 k(\log N + \log k \log^* N)) \ .$$

The number of registers follows from Theorem 1:

$$\mathcal{O}\left(\sum_{j=1}^{\lceil \lg n \rceil} 2^j \log \frac{N}{2^j} \right) = \mathcal{O}\left(n \log \frac{N}{n} \right) \ ,$$

since $k \leq n$. □

The renaming algorithms for the case when N is known use few auxiliary registers if N is polynomial in n.

Corollary 2 *If a known range of the original names N is polynomial in n, then algorithm* RANGE-RENAME(N) *runs in $\mathcal{O}(\log^3 n \log^* n)$ local steps and uses $\mathcal{O}(n \log n)$ auxiliary registers. If a known range of the original names N satisfies $N = \mathcal{O}(n)$, then algorithm* RANGE-RENAME(N) *uses $\mathcal{O}(n)$ auxiliary registers.*

Proof: Algorithm RANGE-RENAME(N) runs in $\mathcal{O}(\log^2 k(\log N + \log k \log^* N))$ local steps, by Theorem 2. The time bound becomes $\mathcal{O}(\log^3 n \log^* n)$ if N is polynomial in n, since $k \leq n$. Algorithm RANGE-RENAME(N) uses $\mathcal{O}(n \log \frac{N}{n})$ auxiliary registers, by Theorem 2. If $N = \mathcal{O}(n)$ then the number of registers is $\mathcal{O}(n \log \frac{N}{n}) = \mathcal{O}(n)$, and if $N = \mathcal{O}(n^\alpha)$, for $\alpha \geq 1$, then the number of registers is $\mathcal{O}(n \log \frac{N}{n}) = \mathcal{O}(n \log n)$. □

Renaming when k is known while N is not. We design an algorithm Square-Rename(k), where a range N is unspecified. The algorithm assigns new names with $2k-1$ as a range of new names.

The algorithm refers to three other algorithms, some for the case when both parameters k and N are known. Let MA(k) be an adaptive algorithm given by Moir and Anderson [45], which is k-renaming with $M = \mathcal{O}(k^2)$ as a bound on new names. It operates in $\mathcal{O}(k)$ local steps and uses $\mathcal{O}(k^2)$ auxiliary registers. Let AF(k, N) be the algorithm of Attiya and Fouren [18], which is (k, N)-renaming with $2k-1$ as a bound on new names. It operates in $\mathcal{O}(N)$ local steps and uses $\mathcal{O}(N^2)$ auxiliary registers. We use algorithm Compact-Rename together with algorithms AF and MA to obtain a new algorithm called Square-Rename(k), which is k-renaming with $2k-1$ as a bound on new names, for any k and N.

Algorithm Square-Rename(k) is structured into three parts. The sets of registers used in all three parts are disjoint. First run algorithm MA(k) to rename, with Ck^2 as a bound on new names, for some $C > 0$. Continue by invoking Compact-Rename(k, Ck^2) to rename again, with $2^{15}k$ as a range of new names, by Theorem 1. The processes execute Compact-Rename(k, Ck^2) with the names obtained in the preceding execution of MA(k) treated as original names. Finally, execute AF($k, 2^{15}k$) to rename one more time. The processes execute AF($k, 2^{15}k$) with names obtained in Compact-Rename(k, Ck^2) treated as original names. The final new names are as assigned by AF($k, 2^{15}k$).

Theorem 3 *Algorithm* Square-Rename(k) *is k-renaming with $2k-1$ as a bound on new names. It operates in $\mathcal{O}(k)$ local steps and uses $\mathcal{O}(k^2)$ auxiliary registers.*

Proof: We rely on the properties of algorithm MA from [45], and on the properties of algorithm AF given in [18]. Correctness follows from the fact that each of the three renaming algorithms properly handles original names, possibly yielded by a preceding execution, assuming they are given correct ranges of original names, if they rely on this information. The range of names is reduced first to Ck^2 by algorithm MA, for a suitable $C > 0$. Then algorithm Compact-Rename takes over, with $N = Ck^2$, and reduces the range of names to $2^{15}k$, by Theorem 1. Finally, algorithm AF reduces the range of names to $2k-1$, while using $N = 2^{15}k$.

The local step complexity of algorithm Square-Rename(k) is

$$\mathcal{O}\left(k + \log k \left(\log k^2 + \log k \log^* \frac{k^2}{k}\right) + k\right) = \mathcal{O}(k) \ ,$$

by the properties of algorithm MA in [45], and by these of algorithm AF given in [18], and by Theorem 1. The number of needed registers is at most

$$\mathcal{O}\left(k^2 + k \log \frac{k^2}{k} + k^2\right) = \mathcal{O}(k^2) \ ,$$

by the respective properties of algorithms MA(k) and AF($k, 2^{15}k$), and by Theorem 1. □

Adaptive renaming with both k and N unknown. Now we develop algorithm Adaptive-Rename solving Renaming in a fully adaptive fashion. The algorithm uses Square-Rename as a subroutine. It operates as follows. A process p executes instantiations of algorithm Square-Rename(2^j), for consecutive integers $i = 0, 1, 2, \ldots$. If p does not obtain a new name in an execution of Square-Rename(2^j), then it proceeds to execute Square-Rename(2^{j+1}). Once a process p acquires a new

name, then it halts. Executions of SQUARE-RENAME(2^j) for different values of j use dedicated sets of auxiliary registers assigned to each possible integer value $j \leq \lceil \lg k \rceil$ and also dedicated ranges of names assigned to j. These ranges are disjoint, and a range for $j+1$ immediately follows a range for j, such that the range for i such that $0 \leq i \leq j$ fill a contiguous segment. Algorithm ADAPTIVE-RENAME does not have the parameters k and N in its code.

Theorem 4 *Algorithm* ADAPTIVE-RENAME *solves Renaming in an adaptive manner. The range of new names is* $M = 8k - \lg k - 1$. *The number of local steps is* $\mathcal{O}(k)$ *and the number of auxiliary registers is* $\mathcal{O}(n^2)$.

Proof: Consider an instantiation of SQUARE-RENAME(2^j) for $j = \lceil \lg k \rceil$, which is the latest possible. At most k processes participate in this execution. By Theorem 3, the magnitude of new names is bounded above by

$$\sum_{i=0}^{\lceil \lg k \rceil} (2^{i+1} - 1) = 2^{\lceil \lg k \rceil + 2} - (\lceil \lg k \rceil + 1) \leq 8k - \lg k - 1 \ .$$

The number of local steps of each process is bounded from above by

$$\mathcal{O}\Big(\sum_{i=1}^{\lceil \lg k \rceil} 2^i \Big) = \mathcal{O}(2^{\lceil \lg k \rceil}) = \mathcal{O}(k) \ ,$$

by Theorem 3. The number of needed auxiliary registers used is

$$\mathcal{O}\Big(\sum_{i=1}^{\lceil \lg k \rceil} 2^{2i} \Big) = \mathcal{O}(2^{2\lceil \lg k \rceil}) = \mathcal{O}(n^2) \ ,$$

again by Theorem 3. □

There is an alternative adaptive solution for Renaming, which works as follows. First execute an adaptive version of algorithm MA. It accomplishes renaming in $\mathcal{O}(k)$ local steps and k^2 new names using $\mathcal{O}(n^2)$ registers. Follow by executing algorithm RANGE-RENAME(k^2). By Theorem 2, this solves Renaming in $\mathcal{O}(k + \log^3 k) = \mathcal{O}(k)$ local steps while using $\mathcal{O}(n^2 + n \log n) = \mathcal{O}(n^2)$ registers. A drawback of this algorithm is that the range of new names, although still $\mathcal{O}(k)$, has a large constant factor by k. This is not the case for the algorithm ADAPTIVE-RENAME, where the range of names is smaller than $8k - \lg k$.

Solutions for Store&Collect. We show how to implement the `Store` and `Collect` operations, with k processes out of n participating, with original names in the range $[N]$. We want the first `Store` by a process to begin by executing a suitable renaming algorithm. A new name identifies a register which the process uses to store by writing into it. Each of the subsequent calls of `Store` takes a constant number of local steps, because the register to write to has already been identified The algorithms for Store&Collect are categorized by the levels of knowledge of k and N, with n always known.

A number M is a range of new names in a renaming algorithm. If M can be computed in advance, based on the available knowledge of the numbers k and N, then an algorithm can be structured as follows. We allocate M shared read-write registers S_i indexed by natural numbers i

in $[M]$. In order to store a value, a process first seeks a new name x in $[M]$. This assigns a unique read-write register S_x. The process with a new name x stores its values by writing into S_x. Collecting is performed by reading the registers S_i, for all $i \in [M]$. This approach works when both parameters k and N are known.

Theorem 5 *If both k and N are known, then Store&Collect can be implemented such that a first storing operation by a process takes $\mathcal{O}(\log k(\log N + \log k \log^* \frac{N}{k}))$ steps and collecting $\mathcal{O}(k)$ steps, while using $\mathcal{O}(k \log \frac{N}{k})$ auxiliary registers.*

Proof: Since we use algorithm Compact-Rename(k, N) for renaming, it is Theorem 1 that summarizes the performance characteristics of renaming. The number of registers to store values corresponding to the new names is $M = 2^{15}k$. The first `Store` operation of a process is preceded by acquiring a new name. This takes $\mathcal{O}(\log k(\log N + \log k \log^* \frac{N}{k}))$ local steps. The `Collect` operation consists of reading all the $2^{15}k$ shared registers, which takes time $\mathcal{O}(k)$. The algorithm uses $\mathcal{O}(k \log \frac{N}{k})$ registers for renaming and $2^{15}k$ registers for storing, which together make $\mathcal{O}(k \log \frac{N}{k})$ auxiliary registers. □

The knowledge of k and a range of original names N that is polynomial in n allows to obtain a Store&Collect algorithm efficient with respect to the total number of processes n.

Corollary 3 *If both k and N are known and $N = \mathcal{O}(n)$ then Store&Collect can be implemented such that a first storing operation by a process takes $\mathcal{O}(\log^2 n)$ local steps and collecting $\mathcal{O}(k)$ steps, while using $\mathcal{O}(n)$ auxiliary registers. If both k and N are known and N is polynomial in n then Store&Collect can be implemented such that a first storing operation by a process takes $\mathcal{O}(\log^2 n \log^* n)$ local steps and collecting $\mathcal{O}(k)$ steps, while using $\mathcal{O}(n \log n)$ auxiliary registers.*

Proof: This is a specialization of the solution of Store&Collect when both k and N are known with performance summarized in Theorem 5. Since we use algorithm Compact-Rename(k, N) for renaming, we refer to Corollary 1 that summarizes it performance characteristics with additional assumptions on the magnitude of N. □

Next, we consider the case when N is known but k is unknown. These assumptions qualify algorithm Range-Rename(N) for renaming to be applicable. This algorithm assigns new names that are in the range between 1 and $2^{16}k$, if k processes contend for new names, by Theorem 2. A solution to Store&Collect needs to have at least $2^{16}n$ registers allocated in advance, since the unknown k could be as large as n.

In order to read only the registers actually used for storing, while performing `Collect`, we suitably mark an initial range of registers during writing into them not to waist time reading unused registers. This is implemented as follows. The number $2^{16}n$ of registers needs to be incremented by $16 + \lceil \lg n \rceil$. These many registers are identified by the integers in the range $[1, 2^{16}n + \lceil \lg n \rceil + 16]$ as their *primary addresses*, in that the ith register has primary address i. All these registers are initialized to be null as usual. The registers with primary addresses of the form $i + 2^i$, for $i \geq 0$, play an *auxiliary* role. If a value written in such a register is different from null then the register is *marked*. The registers with primary addresses between $2^i + i + 1$ up to $2^{i+1} + i$ have secondary addresses between 2^i and $2^{i+1} - 1$, for $i \geq 0$. Once can verify directly that each register is either auxiliary or it has a unique secondary address, and each integer between 1 and $2^{16}n$ is a secondary address of a unique register.

A process with a new name m stores its proposed value along with its original name in a register of the secondary address m. During its first operation `Store`, the process marks all the auxiliary registers whose primary addresses are less than the primary address of the register with m as the secondary address. In order to perform `Collect`, a process reads the consecutive registers up to the first unmarked auxiliary register or all the registers if all auxiliary registers are marked. An unmarked auxiliary register indicates that no value has been stored beyond this primary address.

Theorem 6 *If N is known but k is not, then Store&Collect can be implemented such that a first instance of storing takes $\mathcal{O}(\log^2 k(\log N + \log k \log^* N))$ local steps and collecting $\mathcal{O}(k)$ steps, with $\mathcal{O}(n \log \frac{N}{n})$ auxiliary registers.*

Proof: As a renaming subroutine, we use Range-Rename(N), and refer to Theorem 2, which summarizes the performance characteristics of the algorithm. The number of registers to store values corresponding to the new names is $2^{16}n$, because the range of new names is $M = 2^{16}k$ with $k \leq n$. The first `Store` operation of a process takes time $\mathcal{O}(\log^2 k(\log N + \log k \log^* N))$ to identify a new name and then up to $1 + \lg k$ steps to mark auxiliary registers and store a proposed value. Each of the subsequent `Store` operations takes constant time. The `Collect` operation consists of reading up to $\lceil \lg k \rceil$ auxiliary registers and up to the $2^{16}k$ registers with the smallest secondary addresses. The algorithm uses $\mathcal{O}(n \log \frac{N}{n})$ auxiliary registers for renaming and $2^{16}n + \lceil \lg n \rceil + 16$ registers for storing, which together make $\mathcal{O}(n \log \frac{N}{n})$ registers. □

The algorithms for Store&Collect for the case when N is known use few auxiliary registers if N is polynomial in n.

Corollary 4 *If a known range of the original names N is such that $N = \mathcal{O}(n)$, then Store&Collect can be implemented using $\mathcal{O}(n)$ auxiliary registers. If a known range of the original names N is polynomial in n, then Store&Collect can be implemented using $\mathcal{O}(n \log n)$ auxiliary registers. In each of these cases, a first operation of storing can be performed in a number of local steps polylogarithmic in n and collecting takes $\mathcal{O}(k)$ steps.*

Proof: This is a specialization of the solution of Store&Collect when N is known but k is not with performance summarized in Theorem 6. Since we use algorithm Range-Rename(N) for renaming, we refer to Corollary 2 that summarizes it performance characteristics with additional assumptions on the magnitude of N. □

Next we consider the case when k is known but N is not. We want to use the algorithm Square-Rename(k) for renaming. By Theorem 3, the range of new names can be determined as $M = 2k - 1$ and renaming requires $\mathcal{O}(k^2)$ auxiliary registers. A solution to Store&Collect needs Ck^2 shared registers for renaming, for a suitable $C > 0$, and $2k - 1$ registers for storing.

Theorem 7 *If the number of participating processes k is known but the range of original names N is not, then Store&Collect can be implemented such that a first instance of storing takes $\mathcal{O}(k)$ local steps and collecting $\mathcal{O}(k)$ steps, with $\mathcal{O}(k^2)$ auxiliary registers.*

Proof: The design of the implementations of `Store` and `Collect` follows the general approach of having a dedicated block of $2k-1$ registers for storing and each participating process first acquiring

a new name to identifies a register. By Theorem 3, this takes $\mathcal{O}(k)$ local steps while using $\mathcal{O}(k^2)$ auxiliary registers. To collect the proposed values to build a view, a process reads all $2k-1$ registers used for storing, which takes $\mathcal{O}(k)$ local steps. □

Next we consider the adaptive case with both k and N unknown. We want to use algorithm Adaptive-Rename for renaming. According to its performance characteristics summarized in Theorem 4, the range of new names is $M = 8k - \lg k - 1$ and the number of auxiliary registers is $\mathcal{O}(n^2)$. Registers used for storing can be handled similarly as in the case of unknown k and known N, by arranging a suitably large segment of registers and using them by referring to primary and secondary addresses. We need $8n - \lg n - 1$ secondary addresses, so a block of registers with $8n - \lg n - 1 + \lceil \lg(8n - \lg n - 1) \rceil$ primary addresses suffices. Additionally, we need to allocate Dn^2 registers for renaming, for a suitable $D > 0$.

Theorem 8 *If k and N are both unknown, then Store&Collect can be implemented adaptively such that first storing takes $\mathcal{O}(k)$ steps and collecting takes $\mathcal{O}(k)$ steps, with $\mathcal{O}(n^2)$ auxiliary registers.*

Proof: We use Adaptive-Rename for renaming purposes. The total number of auxiliary registers is $\mathcal{O}(n^2)$, with $\mathcal{O}(n)$ for storing and $\mathcal{O}(n^2)$ for renaming. The first store operation of a process takes $\mathcal{O}(k + \log k) = \mathcal{O}(k)$ local steps, by Theorem 4. The subsequent store operations are of constant time each. The collecting operation takes $\mathcal{O}(k)$ steps, since there is only a prefix of $\mathcal{O}(k)$ registers corresponding to the names to be read. □

An adaptive solution of Store&Collect with performance as in Theorem 8 has already been given by Afek and De Levie [4], by using a different approach.

4 Lower Bounds on Local Steps

We consider the time complexity of renaming and storing for the first time. A *configuration* of the system consists of events enabled by each process, which are either reads or writes. An *execution* consists of events as they occur in time. An event that occurs is selected from a current configuration. A process participating at a current event immediately enables either a read or a write to contribute an option for such a read or write to occur in the next configuration.

A lower bound for renaming. For a distributed system of n processes, an instance of Renaming is determined by some of the following four parameters: an upper bound on the number of participating processes k, a range of new names M, a range of original names N, and a number of auxiliary shared read-write registers r. For some configurations of these parameters, the number of steps can be very small; for instance, if $N = \Theta(k)$ then the original names could do, so the number of steps is $\mathcal{O}(1)$. On the other side of the spectrum, if N is not known and can be arbitrarily large, then the step complexity of any renaming algorithm is $k - 1$.

An intuition why the number of steps is $k - 1$ in some scenarios could be build as follows. If the original names affect the actions of processes, then there is such an assignment of the n original names that at any point of an execution, if there is a choice that processes can make which is affected by the original names, then all the processes might choose similarly. In particular, if processes choose not to write, so that there is no communication among them, then all want to assign the same name. Therefore, one of the processes writes at some point, and let p be the first

such a process. After a write, some processes learn of the value written, by reading. All processes have to learn the value written by p, since otherwise one process among these that never learn could choose the same name as p. This argument can be extended by induction to imply that the process that chooses the name as the last one had to read at least $k-1$ times. A general lower bound of Jayanti et al. [41] captures related scenarios.

We present a lower bound which gives an estimate on N, depending on fixed range of original names M and the number of auxiliary shared read-write register r, for which $k-1$ steps during assigning new names are necessary. The lower bound is flexible enough to be applicable to scenarios when algorithms of poly-logarithmic step complexity exist.

Theorem 9 *Let k processes among a total of n in an asynchronous system using r shared read-write registers and with original names in the range $[N]$ execute a wait-free renaming algorithm that assigns new names in a range $[M]$. Then there exists an execution in which some process performs $1+\min\{k-2, \lfloor \log_{2r} \frac{N}{M+k-1} \rfloor\}$ steps.*

Proof: Consider a conceptual set consisting of N processes, each identified by a different original name. The first goal is to identify a subset K of these processes consisting of at most k elements so that an execution of the algorithm by these processes results in a necessary number of local steps. The construction of the set K is recursive and proceeds through a sequence of stages.

A stage represents a group of concurrent reads or writes to shared registers. A stage determines groups of processes we call *pool* and *residue*. When the stages get completed, the pool and the residue together make a set K we seek. In notation, let *stage* i result in determining a *pool set* $P(i+1)$ of processes eligible to be considered for stage $i+1$, a *residue set* $Q(i+1)$, by determining a prefix $\mathcal{E}(i)$ of an execution. The construction starts with the initial pool $P(0)$ consisting of N conceptual processes, each identified by its original name, the initial residue $Q(0)$ is empty, and $\mathcal{E}(0)$ is an empty prefix of an execution.

Suppose we have a configuration with a determined sets $P(i)$ and $Q(i)$ and a prefix $\mathcal{E}(i)$ of the execution. Let $R(i)$ consist of the processes in $P(i)$ that have a read enabled in the configuration and $W(i)$ consist of the processes in $P(i)$ that have a write enabled. There are two cases we considered next that determine $P(i+1)$, $Q(i+1)$, and $\mathcal{E}(i+1)$.

Suppose first that $|R(i)| \geq |W(i)|$. By the pigeonhole principle, there is a register which a group of at least these many processes want to read:

$$\frac{|R(i)|}{r} \geq \frac{|P(i)|}{2r} \ .$$

Define the pool $P(i+1)$ to be this group of processes and the residue $Q(i+1)$ to be equal to $Q(i)$. The prefix $\mathcal{E}(i+1)$ is obtained from $\mathcal{E}(i)$ by having the processes in $R(i)$ read the register in arbitrary order.

Suppose next that $|R(i)| < |W(i)|$. By the pigeonhole principle, there is a register to which a group of at least these many processes want to write to:

$$\frac{|W(i)|}{r} \geq \frac{|P(i)|}{2r} \ .$$

Define the pool $P(i+1)$ to be this group of processes. The prefix $\mathcal{E}(i+1)$ is obtained from $\mathcal{E}(i)$ by having the processes in $W(i)$ write to the register in arbitrary order. A process p that writes last in this group is added to $Q(i)$ to obtain $Q(i+1) = Q(i) \cup \{p\}$.

As the recursive construction progresses, for $i \geq 0$, the following invariant is maintained:

1) the pool $P(i)$ includes at least $\frac{N}{(2r)^i}$ processes,

2) all the processes in $P(i)$ have exactly the same history in $\mathcal{E}(i)$ of reading from shared registers,

3) the residue $Q(i)$ includes at most i processes,

4) all the processes that wrote a value to a shared register that has ever been read in $\mathcal{E}(i)$ are in $Q(i)$.

We continue for the maximum number of stages t such that two constraints are met. One is that the inequality $\frac{N}{(2r)^t} \geq M + k - 1$ holds, where M is the range of new names. The resulting stage number t satisfies $t \leq \log_{2r} \frac{N}{M+k-1}$ and $P(t)$ contains at least $M + k - 1$ elements. The other constraint is that $t \leq k - 2$, so that $Q(t)$ has at most $k - 2$ elements. These two requirements combined determine

$$t = \min \left\{ k - 2, \left\lfloor \log_{2r} \frac{N}{M + k - 1} \right\rfloor \right\} .$$

The definitions of $P(t)$ and $Q(t)$ imply that the processes in $P(t) \setminus Q(t)$ have not written in $\mathcal{E}(t)$ yet and have read the same values from the same shared registers in the same order of reading. The set $P(t) \setminus Q(t)$ has at least $M + k - 1 - (k - 2) \geq M + 1$ elements.

Suppose that a decision on a new name is made by each among the processes in $P(t) \setminus Q(t)$ were possible without any further reads or writes. The range of new names has M elements. By the pigeonhole principle, there are two processes p_1 and p_2 in $P(t) \setminus Q(t)$ that would get assigned the same name. This means some process among p_1 and p_2 performs at least one additional read. We set $K = Q(t) \cup \{p_1, p_2\}$, which has at most k elements.

There is an execution $\mathcal{E}$ of the algorithm, with the processes in K as the only contenders for new names, such that $\mathcal{E}(t)$ restricted only to the events involving the processes in K is a prefix of $\mathcal{E}$. The processes that wrote values ever read in $\mathcal{E}(t)$ are in K. The processes p_1 and p_2 have the same history of reads in $\mathcal{E}(t)$, and each of them read t times without writing even once. The algorithm is a wait-free solution to Renaming, so both p_0 and p_1 eventually assign themselves names in $\mathcal{E}$. It follows that at least one of the processes p_1 and p_2 eventually performs at least one read after $\mathcal{E}(t)$. This means that the process makes at least $1 + t$ steps in execution $\mathcal{E}$. □

Theorem 9 implies that some process executing a renaming algorithm has to perform at least $k - 1$ steps in a suitable distributed system, which we show next.

Corollary 5 *For any k-renaming algorithm for processes with new names in some range $[M]$ and using some r shared read-write registers, there exists a sufficiently large range of original names $[N]$ and an execution in which some process performs at least $k - 1$ steps.*

Proof: Let us choose N such that the part $\log_{2r} \frac{N}{M+k-1}$ in the lower bound of Theorem 9 is at least $k - 2$. To thus end, it suffices to set N to be at least as large as $(M + k - 1)(2r)^{k-2}$. Then the lower bound of Theorem 9 becomes $k - 1$. □

Algorithm Square-Rename(k) works for arbitrary range $[N]$ of original names. Corollary 5 implies that this algorithm is asymptotically optimal with respect to local-step performance, which is $\mathcal{O}(k)$.

Corollary 6 *For any (k, N)-renaming algorithm that has $M = \mathcal{O}(k)$ as a bound on new names and uses $\mathcal{O}(k \log \frac{N}{k})$ auxiliary registers, for a range of original names $N = \Omega(n^2)$, there exists an execution in which some process performs $\Omega\big(\frac{\log n}{\log k + \log\log n}\big)$ steps.*

Proof: We reformulate the logarithmic part of the bound in Theorem 9 as follows:

$$\log_{2r} \frac{N}{M+k-1} = \frac{\lg \frac{N}{M+k-1}}{\lg(2r)} = \frac{\lg N - \lg(M+k-1)}{\lg r + 1} \ . \tag{3}$$

By the assumptions, we have $\Omega(1) + 2\lg n \le \lg N$, $\lg(M+k-1) \le \lg(2M) \le \mathcal{O}(1) + \lg k$, and $\lg r \le \lg k + \lg\lg n + \mathcal{O}(1)$. Substitute these into (3) to obtain a bound

$$\frac{\lg N - \lg(M+k-1)}{\lg r + 1} \ge \frac{\lg n + \Omega(1)}{\lg k + \lg\lg n + \mathcal{O}(1)} \ ,$$

which is $\Omega\big(\frac{\log n}{\log k + \log\log n}\big)$. □

Corollary 6 implies that algorithm Compact-Rename(k, N) may miss local-step optimality by a factor that is about $\log^2 k$. More precisely, assuming additionally that $k = \Omega(\log n)$, the lower bound of Theorem 9 becomes $\Omega\big(\frac{\log n}{\log k}\big)$. Each process executing algorithm Compact-Rename(k, N) performs $\mathcal{O}(\log k(\log N + \log k \log^* \frac{N}{k}))$ steps, by Theorem 1. We have the following asymptotic identity $\log^2 k \cdot \Omega\big(\frac{\log n}{\log k}\big) = \Omega(\log k(\log N + \log k \log^* \frac{N}{k}))$, under the additional assumption about k that $\log k \cdot \log^* \frac{N}{k} = \mathcal{O}(\log n)$.

A lower bound on first storing. The lower bound on the first operations of storing applies to implementations of solutions of Store&Collect in which a process proposes a value by writing it to a dedicated register, one such a register per process, and collecting is performed by reading all such registers. The first operation `Store` that a process performs requires identifying a register to write a proposed value. We assume there are r shared registers available. The processes have names from the range $[N]$.

Theorem 10 *Let k processes among a total of n in an asynchronous system using r auxiliary shared read-write registers and with names in the range $[N]$ execute a wait-free algorithm for Store&Collect in a system with r shared read-write registers. Then there exists an execution in which some process performs $1 + \min\{k-2, \lfloor \log_{2r} \frac{N}{r+k-1} \rfloor\}$ steps in its first operation `Store`.*

Proof: A proof is similar to that of Theorem 9. We structure a prefix of an execution $\mathcal{E}$ to reflect stages, with a pool $P(i)$ and residue $Q(i)$ sets of processes after stage i. An execution continues for the maximum number of stages t such that two constraints are met. One is that the inequality $\frac{N}{(2r)^t} \ge r + k - 1$ holds. The resulting stage number t satisfies $t \le \log_{2r} \frac{N}{r+k-1}$ and $P(t)$ contains at least $r + k - 1$ elements. The other constraint is that $t \le k - 2$, so that $Q(t)$ has at most $k - 2$ elements. These two requirements combined determine

$$t = \min\left\{k-2, \left\lfloor \log_{2r} \frac{N}{r+k-1} \right\rfloor\right\} \ .$$

The definitions of $P(t)$ and $Q(t)$ imply that the processes in $P(t) \setminus Q(t)$ have not written in $\mathcal{E}(t)$ yet and have read the same values from the same shared registers in the same order of reading. The set $P(t) \setminus Q(t)$ has at least $r + k - 1 - (k-2) \ge r + 1$ elements.

Suppose that selections of registers for storing by each among the processes in $P(t) \setminus Q(t)$ were possible without any further reads or writes. There are r shared registers. By the pigeonhole principle, there are two processes p_1 and p_2 in $P(t) \setminus Q(t)$ that would select the same register. This means some process among p_1 and p_2 performs at least one additional read. We set $K = Q(t) \cup \{p_1, p_2\}$, which has at most k elements.

There is an execution $\mathcal{E}$ of the algorithm, with the processes in K as the only competitors to select registers for storing, such that $\mathcal{E}(t)$ restricted only to the events involving the processes in K is a prefix of $\mathcal{E}$. The processes that wrote values ever read in $\mathcal{E}(t)$ are in K. The processes p_1 and p_2 have the same history of reads in $\mathcal{E}(t)$, and each of them read t times without writing even once. At least one of the processes p_1 and p_2 eventually performs at least one read after $\mathcal{E}(t)$. This means that the process makes at least $1 + t$ steps in execution $\mathcal{E}$. □

Theorem 10 implies that for each solution of Store&Collect some process has to perform at least $k - 1$ steps during its first storing, in a suitable distributed system, which we show next.

Corollary 7 *For any Store&Collect algorithm for a system with some r shared read-write registers, there exists a sufficiently large range of original names $[N]$ and an execution in which some process performs at least $k - 1$ steps.*

Proof: Let us choose N such that the part $\log_{2r} \frac{N}{r+k-1}$ in the lower bound of Theorem 10 is at least $k - 2$. To thus end, it suffices to set N to be at least as large as $(r + k - 1)(2r)^{k-2}$. Then the lower bound of Theorem 10 becomes $k - 1$. □

The algorithm for Store&Collect with performance characteristics summarized in Theorem 7 works for an arbitrary range $[N]$ of original names. Corollary 7 implies that this algorithm is asymptotically optimal with respect to local-step performance of first storing, which is $\mathcal{O}(k)$.

Corollary 8 *For any algorithm for Store&Collect that uses $r = \mathcal{O}(k \log \frac{N}{k})$ auxiliary registers for a range of original names $N = \Omega(n^3)$, there exists an execution in which some process performs $\Omega\big(\frac{\log n}{\log k + \log\log n}\big)$ steps in its first storing.*

Proof: Reformulate the logarithmic part of the bound in Theorem 10 as follows:

$$\log_{2r} \frac{N}{r+k-1} = \frac{\lg \frac{N}{r+k-1}}{\lg(2r)} = \frac{\lg N - \lg(r+k-1)}{\lg r + 1} \,. \tag{4}$$

By the assumptions, we have $\Omega(1) + 3 \lg n \le \lg N$ and $\lg r \le \lg(r + k - 1) \le \lg k + \lg\lg n + \mathcal{O}(1)$. Substitute these into (3) to obtain a bound

$$\frac{\lg N - \lg(r+k-1)}{\lg r + 1} \ge \frac{\lg n + \Omega(1)}{\lg k + \lg\lg n + \mathcal{O}(1)} \,,$$

which is $\Omega\big(\frac{\log n}{\log k + \log\log n}\big)$. □

Corollary 8 implies that the algorithm for Store&Collect for the case when both k and N are known, and whose performance characteristics are given in Theorem 5, may miss local-step optimality of first storing by a factor that is about $\log^2 k$. More precisely, assuming additionally that $k = \Omega(\log n)$, the lower bound of Theorem 10 becomes $\Omega\big(\frac{\log n}{\log k}\big)$. Each process storing for the first

time performs $\mathcal{O}(\log k(\log N + \log k \log^* \frac{N}{k}))$ steps, by Theorem 5. We have the following asymptotic identity $\log^2 k \cdot \Omega\big(\frac{\log n}{\log k}\big) = \Omega(\log k(\log N + \log k \log^* \frac{N}{k}))$, under the additional assumption about k that $\log k \cdot \log^* \frac{N}{k} = \mathcal{O}(\log n)$.

5 Unbounded Selection

We consider problems concerning processes selecting positive integers continuously such that each selection is exclusive. A selected positive integer may be considered as an abstract name from an unbounded range. Such names can be used to identify registers from an infinite pool of registers. Infinitely repeated selections of names are efficient when they minimize the number of positive integers that never get selected to be assigned as names.

Next, we review the functionality of a distributed dynamic data structure that we call a "repository." We will refer to two related concepts of "storing" and "depositing" a value in a register. A value written to a register gets stored in it as long as it is not overwritten by a different value. Depositing a value means storing it indefinitely in some register. A repository is a concurrent data structure for depositing values in shared read-write registers. The underlying assumption is that, for each point in time, each process will eventually generate a value to be deposited in a repository.

We assume that there are infinitely many shared read-write registers, denoted $R[1], R[2], R[3], \ldots$, used to store deposited values. We say that these registers are *dedicated* to depositing and that i in the *index of register* $R[i]$. We assume random access to these shared registers, in that an index i allows to identify the shared register $R[i]$ and perform a read or write on it. Every register $R[i]$ is initialized to `null`, which is interpreted as the register being empty. An algorithm managing repeated deposits may also use additional auxiliary registers.

A formal definition of depositing and a repository refers to an algorithm implementing this operation and supporting registers. Consider an execution of an algorithm implementing depositing. A value x is considered *deposited in a register* R at an event in the execution, when the following is satisfied in the system's configuration just after the event:

1. The value x is stored in register R.
2. The value x will not be overwritten in register R by any different value in the following events of the execution.

A repository is a distributed data structure that provides a repertoire of operations for each participating process. We describe these operations next.

A process p may invoke an operation $\texttt{Query}_p$ to obtain a new value to be deposited. An event $\texttt{Return}_p(v)$ returns a value v for $\texttt{Query}_p$, where $v \neq \texttt{null}$ is a value to deposit. An event $\texttt{Return}_p(\texttt{null})$ indicates that there is no value to deposit yet.

A process p invokes an operation $\texttt{Deposit}_p(v)$ to deposit a value $v \neq \texttt{null}$. An event $\texttt{Ack}_p(i)$ acknowledges completing $\texttt{Deposit}_p()$, where $R[i]$ is the register in which the value has been deposited. When a process crashes while working to deposit a value, and depositing this value has not been acknowledged before the crash, then the value may either get deposited or not in a dedicated register.

Following the standard understanding of simulating executions, as presented by Attiya and Welch [24], we prohibit "pipelining" on the operations `Query` and `Deposit`. This means that a process may invoke a new operation only after the previous one, if any, has returned an outcome or has been acknowledged as completed. When a process p obtains a return of a query for a new value, then the process eventually invokes $\texttt{Query}_p$ again. We assume *fair occurrence of deposit requests* at processes, which means that each process eventually obtains a new value to deposit, after having deposited the previous value, if any, unless the process crashes.

Let us observe that no algorithm depositing values and resilient to even one crash can guarantee for a specific register that a value gets deposited in the register eventually, since otherwise the value stored there could be used to determine a decision in a solution of Consensus, which is impossible in asynchronous systems with processes prone to crashes and read-write registers [24, 39, 42]. This means that for any execution of a depositing algorithm, some registers dedicated for depositing may never be used for deposits. Now the question is how many registers dedicated for depositing will never be used to deposit? There is a simple solution to the problem of repeated deposits in which a process i deposits only in consecutive registers with indices congruent to i modulo n. The problem with this approach is that if even one process crashes then infinitely many registers dedicated for depositing will never deposit a value. We want to have a solution to repeated depositing in which the number of dedicated registers not used for deposits is finite in any infinite execution.

A *repository* is a concurrent data structure that allows each process to deposit values in dedicated registers that satisfied the following two properties, subject to fair occurrence of deposit requests in an infinite execution:

Persistence: Starting from an acknowledgment event $\texttt{Ack}_p(i)$, for a process p and register $R[i]$ dedicated for depositing, the value stored in $R[i]$ by p becomes deposited.

Utilization: There are finitely many registers dedicated for depositing that never store a deposited value.

We want the quality of an implementation of a repository to be be minimally non-blocking, but preferably wait-free. These qualities have the following standard meaning:

Non-blocking: If at least one nonfaulty process wants to deposit a value in a configuration, then eventually a value gets deposited.

Wait-free: If some nonfaulty process wants to deposit a value in a configuration, then eventually this process succeeds in depositing its value.

The Repository problem is to implement a repository in a distributed system of n processes prone to crashes and an unbounded supply of read-write registers initialized to `null`.

Implementations of a repository. The algorithms we give next assign integers to processes such that an assignment provides exclusivity of a selection of an integer by a process. We interpret a newly assigned integer i as an indication that the register $R[i]$ could be available for depositing.

We start from the algorithm called Selfish-Repository, which works as follows. Each process p maintains a sorted list L_p of $2n-1$ indices i in its local memory, which are interpreted as identifying registers possibly still available for deposits. The list is initialized to store the first $2n-1$ positive integers arranged in order in consecutive entries. Pointer $L_p.\texttt{head}$ points to the first

item on the list. For an entry x in the list, $x.\texttt{next}$ is the next item in the list, and $x.\texttt{value}$ is the integer stored in the entry. Process p also uses a variable A_p interpreted as an index of the next available empty register immediately after the registers whose indices are in L_p. The variable A_p is initialized to $2n$. The entry at position i in list L_p of process p is denoted $L_p[i]$ (or $L[i]$ when p is understood).

Process p may *update list* L_p, which is performed as follows. Process p scans L_p, starting from $L_p.\texttt{head}$. For each entry x scanned in the list, if $x.\texttt{value} = j$, then p reads $R[j]$. If $R[j]$ is still empty, then the next entry $x.\texttt{next}$ is considered, otherwise p removes entry x from list L_p, by making the predecessor of x point to its successor $x.\texttt{next}$, and begins scanning registers $R[i]$ one by one in the order of indices, starting from the index i stored in A_p. The scan of array R continues until an empty $R[k]$ is found, if any. Once process p reads an empty $R[k]$ then p appends a new entry y with $y.\texttt{value} = k$ to L_p, and sets A_p to $k+1$. This completes processing entry x. Next the entry $x.\texttt{next}$ in L_p is processed in a similar manner. Updating list L_p ends after all the entries of L_p have been processed.

We will use an atomic-snapshot object $\mathcal{S}$. It includes read-write registers $S[p]$, for each process p, for $1 \leq p \leq n$, such that $S[p]$ is writable by p and readable by all processes. A view returned to a process that invoked taking a snapshot consists of a vector $V = (V[1], V[2], \ldots, V[n])$, where $V[i]$ stores the value read from $S[i]$. Each register $S[i]$, for $1 \leq i \leq n$, is initialized to $\texttt{null}$. The registers $S[i]$ are supported by other auxiliary registers in $\mathcal{S}$ to equip S with the functionality of an atomic snapshot object, see [1]. Process p writes an integer in the interval $[1, 2n-1]$ to the register $S[p]$ in $\mathcal{S}$. After taking a snapshot using $\mathcal{S}$, process p assigns itself the *rank* defined as follows: it is the rank of p among the indices q such that the variable $V[q]$ stores an entry different from $\texttt{null}$. A snapshot determines integers in the interval $[1, 2n-1]$ that are *available*: these are the numbers in the interval that do not occur in the snapshot. For each snapshot with at least one repetition of entries there are always at least n integers available: this is because at most $n-1$ numbers in $[1, 2n-1]$ occur in the snapshot.

The need for a deposit occurs when a process p that has been querying for a new value to deposit obtains such a value. Then process p begins attempts to acquire an index of a register available for deposits. Such attempts are performed repeatedly until identifying an index of a register i exclusively available to p, we call such i a *list name*. Each attempt begins with p reviewing its list L_p and setting the *candidate* index, of an eligible register dedicated to depositing, to $L_p[1]$. The first goal is to go through a sequence of candidates to eventually identify a list name, while verifying each candidate until it passes the verification and becomes a list name.

Identifying a candidate and verification proceed as follows. After p sets the candidate to $L[1]$, it repeats the following three actions. First p sets $S[p]$ to the candidate index. Then p invokes obtaining a view V from the snapshot object $\mathcal{S}$. The third action depends on whether the candidate is unique in the view V. If this is the case then p treats the candidate index in L as a list name produced by the list. Otherwise, p sets r to the rank of p in V, then sets j to the rth integer available in $[1..2n-1]$, and finally chooses a next candidate as the entry on L at position j. Now p resumes verifying the candidate. When process p acquires a list name j then it checks to see if $R[j]$ is empty. If this is the case then p deposits at $R[j]$. Otherwise, when $R[j]$ already stores a deposited value then p resumes attempts to acquire an index of an eligible register.

A pseudocode of algorithm Selfish-Deposits is presented in Figure 2. In the pseudocode, the names of variables `candidate` and `list-name` half self-explanatory meaning. The pseudocode

algorithm SELFISH-REPOSITORY

1. `repeat`
   (a) update list L; `list-name` $\leftarrow$ `null`; `candidate` $\leftarrow L[1]$;
   (b) `repeat`
      i. $S[p] \leftarrow$ `candidate`
      ii. obtain a view V from snapshot object $\mathcal{S}$
      iii. `if candidate` is unique in the view V `then list-name` $\leftarrow$ `candidate else`
         A. $r \leftarrow$ the rank of p in V
         B. $j \leftarrow$ the rth integer available in $[1..2n-1]$
         C. `candidate` $\leftarrow L[j]$
      `until list-name` $\neq$ `null`
   (c) `if` R[`list-name`] = `null then`
      i. R[`list-name`] $\leftarrow v$
      ii. $S[p] \leftarrow$ `null`
      iii. `return Ack(list-name)`

Figure 2: Pseudocode of procedure `Deposit`(v), for a process p, implementing a selfish repository.

is structured as a repeat loop (1.), which terminates by return of acknowledgement in the last line (1(c)iii).

Theorem 11 *Depositing based on algorithm* SELFISH-DEPOSIT *is a non-blocking implementation of a repository such that in each execution at most* $2n-2$ *dedicated deposit registers are not used for depositing.*

Proof: When a value v gets stored in a register R[`list-name`] by a process p in instruction (1(c)i) in Figure 2, then this occurs after `list-name` has been verified to be unique in the view returned by the snapshot object. This entry written by p in $S[p]$ stayed there until after p completed writing to R[`list-name`]. This means that at most one process could attempt to store a value in this register R[`list-name`]. The first such process would succeed, as all subsequent attempts, if any, would be prevented by the verification in line (1b) of the pseudocode. This provides the property of Persistence defining a repository.

Next we argue that there are infinitely many successful deposits, assuming fair occurrence of deposit requests. Suppose there is an event after which no deposits occur. Eventually every process either has crashed or it has a value to deposit, by the assumed fair occurrence of deposit requests. Each failure to deposit starts a new iteration of the main repeat loop in Figure 2, which begins with updating list L by instruction (1a). As all the non-faulty processes keep updating the lists, while no deposits occur, then eventually all their lists become equal and store the indices of the smallest empty deposit registers. The values on this list make a set of $2n-1$ natural numbers. Let us take the first event when this occurs. Consider the first following event when each non-faulty process p has written to $S[p]$. Starting from this point in the execution, the ranks of all the processes become fixed in the snapshot object. Consider the subsequent writes to $S[p]$ by a process p. If such a write does not produce a unique number in the view, then each next write of a proposed list name is after choosing by rank. Once the ranks become fixed, each choosing by rank produces a unique entry in the common list L shared by all the processes. It follows that eventually some non-faulty process p

acquires a list name. Now this list name identifies a unique entry in the list L shared by all the processes. The register dedicated for deposits with the index in this unique entry of L is empty and no other process attempts to use this register, so p completes a deposit. This contradicts the assumption that there are no deposits after some event.

If a process crashes in the course of depositing, then the process may have identified a register $R[k]$ for depositing by acquiring a list name k, but the crash occurred before the value got stored in the register $R[k]$. There may be up to $n-1$ such indices k and the corresponding registers $R[k]$. A crashed process p may have set $S[p]$ in the snapshot object to its candidate, which now will store its value $S[p]$ throughout the execution. If such registers $S[p]$ make an initial segment $S[1..k]$, then eventually the first k numbers in lists L stabilize and each of them is a list name assigned to a crashed process. If the next k entries in the lists L stabilize as well, then these will forever stay as the first k available indices never to be assigned as list names. Since there are at most $n-1$ crashes and $k \leq n-1$, we have that up to $n-1$ indices of registers dedicated for deposits may never be used as list names. This means that up to $2n-2$ registers dedicated for depositing may never store a deposited value. □

Theorem 12 *For each non-blocking algorithm implementing a repository there exists an execution in which $n-1$ registers dedicated for depositing remain unused.*

Proof: We argue that in each implementation of a repository, at least $n-1$ dedicated registers may be never used for deposits in some execution. Namely, when a process p is to deposit by writing to a register $R[i]$, and a write event to store the value is enabled, we may "freeze" the write. At this point, no other process q will want to deposit to $R[i]$, because otherwise after $\texttt{ack}_q(i)$ happens, the pending write of p to store at $R[i]$ might occur as well, which results in overwriting $R[i]$, in contradiction of the definition of a repository. This means that if p crashed rather than merely "freezed," then the register R is never used for depositing by any other process. Up to $n-1$ crashes can happen, so at least these many registers might never be used for deposits. □

By Theorem 11, algorithm Selfish-Deposit leaves $\mathcal{O}(n)$ registers dedicated for deposits that remain unused. This combined with Theorem 12 shows that algorithm Selfish-Deposit leaves an asymptotically optimum number of shared registers in the worst case in a perpetual state of not storing a deposited value.

Next, we consider a wait-free implementation of a repository. We call the algorithm that provides the implementation Altruistic-Deposit. The algorithm uses some of the mechanisms in Selfish-Deposit and extends them. The difference between the two algorithms is what a process p does with acquired list names. In algorithm Selfish-Deposit, a process acquiring list names uses it selfishly as address of registers to deposit. In executions of algorithm Altruistic-Deposit, processes share acquired list names with other processes to help in their deposits.

Algorithm Altruistic-Deposit consists of two threads. One auxiliary thread produces register indices, and the other thread deposits values. The two threads are interleaved in a fair manner, in that each process alternates invoking instructions from the two threads. Both threads work on an $n \times n$ array $\texttt{Help}[i,j]$, for $1 \leq i,j \leq n$, of shared read-write registers. The processwa use a snapshot object $\mathcal{S}$ to obtain new list names, similarly as in algorithm Selfish-Deposit. A process i writes a verified list name into $\texttt{Help}[i,j]$ to be used by process j for its deposits.

A process p in the producing thread keeps reading the registers in row $\texttt{Help}[p,*]$ in a round-

algorithm ALTRUISTIC-REPOSITORY : producing thread

```
column ← p
repeat
  1. if Help[p, column] ≠ null then increment column in a round robin manner else
     (a) update list L; list-name ← null; candidate ← L[1];
     (b) repeat
           i. S[p] ← candidate
          ii. obtain a view V from snapshot object S
         iii. if candidate is unique in the view V then list-name ← candidate else
              A. r ← the rank of p in V
              B. j ← the rth integer available in [1..2n − 1]
              C. candidate ← L[j]
         until list-name ≠ null
  2. if R[list-name] = null then
     (a) R[list-name] ← reserved
     (b) Help[p, column] ← list-name
  3. S[p] ← null
```

Figure 3: Pseudocode of a producing thread in the altruistic repository, used by a process p to obtain an index of a register available for depositing. Such an index gets stored as entry $\texttt{Help}[p, \texttt{column}]$ of the column $\texttt{Help}[p, *]$ of the array Help.

robin manner starting from the diagonal entry. If p finds some register $\texttt{Help}[p, c]$ equal to null then p works to obtain a new list name. After successfully acquiring a list name i, process p verifies if $R[i]$ is empty, which means it has not been reserved yet. If this is the case then process p first marks $R[i]$ as reserved and then writes i into $\texttt{Help}[p, c]$. The value reserved is assumed to be different from null. A pseudocode of the producing thread is in Figure 3.

Lemma 9 *For each process p and an event it is involved, eventually some entry in the column $\texttt{Help}[*, p]$ stores an index of a reserved register available for deposits.*

Proof: Processes execute producing threads similarly as depositing selfishly in that this produces new reserved registers in a non-blocking manner. As new entries in the array $\texttt{Help}[*, *]$ get reserved written in them, the writers wrap around their rows in a round robin manner. The perpetual existence of a column of the array $\texttt{Help}[*, p]$ with all entries null would contradict the non-blocking progress achieved in the execution. If a process q obtains a new list-name in an execution of producing thread, then this number is unique in the view provided by the snapshot object. This means that no write to $R[\texttt{list-name}]$ by some other process is pending when q reads $R[\texttt{list-name}]$ and finds it empty in instruction (2) in Figure 3, and so eligible to write reserved. □

A process p in the depositing thread keeps reading the column $\texttt{Help}[*, p]$ in a round-robin fashion, starting from the diagonal entry. Once p finds an index $j \neq \texttt{null}$ stored at $\texttt{Help}[r, p]$, then p deposits in $R[j]$ and then writes null to erase value j in $\texttt{Help}[r, p]$. A pseudocode of the depositing thread is in Figure 4.

algorithm ALTRUISTIC-REPOSITORY : depositing thread

1. `row` $\leftarrow p$
2. `while Help[row`$, p]$ = `null do` increment `row` in a round robin manner
3. `index` $\leftarrow$ `Help[row`$, p]$
4. `Help[row`$, p] \leftarrow$ `null`
5. R[`index`] $\leftarrow v$
6. `Ack(index)`

Figure 4: Pseudocode of a depositing thread, used by a process p to deposit a value v, implementing an altruistic repository.

Theorem 13 *Depositing based on algorithm* ALTRUISTIC-DEPOSIT *is a wait-free implementation of a repository such that at most* $(n+2)(n-1)$ *dedicated deposit registers will never be used for depositing.*

Proof: Consider an event in which a process q wants to deposit a value v. The process invokes the depositing thread and so keeps reading entries in the column `Help`$[*, q]$ in a round-robin manner. By Lemma 9, process q eventually reads `Help`$[r, q] = j$, for some row r and index j. The register $R[j]$ was verified to be empty by the process p that wrote j to `Help`$[r, q]$, which occured when executing line (2) in Figure 3. Process q can safely store the value v in $R[j]$, because while the entry `Help`$[r, q]$ stays equal to index j, register $R[j]$ stays equal to `reserved`, which is different from `null`. This prevents the index j to be written at other locations of the array `Help`, by instruction (2) in Figure 3, and so prevents multiple values to be possibly stored in succession at $R[j]$.

Next, we estimate the number of registers dedicated for depositing that may never be used to deposit a value. This number if maximized when $n-1$ processes crash while many entries of `Help`$[*, *]$ store indices of registers reserved for depositing. Let p be the only process that never crashes. The worst case scenario occurs when each of the crashed processes q has a full column of n indices reserved and it crashes when working to produce an index of a register to be placed in column `Help`$[*, p]$. Such a process q may have written some list name i to $S[q]$ and verified that $R[i]$ = `null` but crashed before setting $R[i]$ to `reserved` and so also did not reset $S[q]$ to `null`. Suppose all the lists L stabilized to the same sequence of entries, and further that such indices i make the first $n-1$ entries in the lists. Then the first $n-1$ entries will never be removed from the lists along with the first $n-1$ entries available in the lists L. The registers with theses $2n-2$ indices will stay equal to `null` forever. We have obtained $n(n-1)$ reserved registers and $2(n-1)$ empty registers never to be used for depositing. The total number of registers dedicated for depositing that never store a deposited value could be $n(n-1) + 2(n-1) = (n+2)(n-1)$. □

It is an open problem if there exists a wait-free implementations of a repository that leaves out $o(n^2)$ registers not used for depositing.

Mining names. Next, we consider the task to have processes work continuously to accumulate a possibly unlimited collection of exclusive names. The distributed system consists of n processes prone to crashes and a number of shared objects. We call designing an algorithm for this task the *Mining-Names* problem. The complete specification is as follows.

A positive integer i is considered to be *assigned to process p as a name* when p exclusively

commits to integer i by writing i in a dedicated local write-once memory variable. Exclusivity means that no two processes ever commit to the same integer, so a name can be interpreted as an exclusive reservation of a natural number. Committing to a name resembles committing to a decision in solutions of Consensus. After committing to a name, a process can proceed to commit to some other natural number as a name as well. A process participating in acquiring new names never stops voluntarily.

An algorithm is a solution to Mining-Names if in each infinite execution the following two properties are satisfied:

Naming: No two different processes ever commit to the same integer as a shared name.

Utilization: There are finitely many positive integers that never get acquired as names.

A mining names solution in a system with process crashes could be non-blocking or wait-free, which is understood as follows.

Non-blocking: For each event in an execution, eventually a new name gets acquired.

Wait-free: For each process and any event in an execution, eventually this process acquires a new name.

Algorithms implementing a repository can be adapted to mining names. Namely, let every process keep invoking an operation to deposit a dummy value v. Rather than deposit v in a register $R[i]$, for some index i, the process commits to the name i. After a new name has been acquired, the process invokes depositing a dummy value again. This transformation from depositing to mining names requires the same distributed system that supports depositing. In particular, the implementation of repository with properties summarized in Theorems 11 and 13 assumes that an infinite array of shared read-write registers each initialized to `null` is available.

Theorem 14 *The Mining-Names problem can be solved by n processes in a non-blocking fashion by an algorithm that leaves at most $2n-2$ nonnegative integers not assigned as names, or in a wait-free manner by an algorithm that leaves at most $(n+2)(n-1)$ integers never assigned as names.*

Proof: This follows from combining the general transformation of implementations of a repository to mining names and Theorems 11 and 13. □

Mining names can be used to implement a repository by a general transformation, which works as follows. Every process keeps mining names, and as soon as a new name i is acquired, this reserves the register $R[i]$ for depositing.

Theorem 15 *For each non-blocking algorithm for Mining-Names, there exists an execution in which $n-1$ positive integers are not assigned as names to any process.*

Proof: We apply the general transformation from an algorithm mining names to an implementaiton of a repository. This transformation creates a non-blocking algorithms mining names from a non-blocking implementation of a repository. If an algorithm for mining names could guarantee fewer than $n-1$ positive integers never assigned as names then this could be converted into a non-blocking solution for a repository that leaves out at most $n-1$ registers never used for deposits. This would contradict Theorem 12. □

The implementations of repository we developed have processes use newly acquired list names as indices of registers in an unbounded array of read-write registers. This works only if a distributed system includes an unbounded array of shared read-write registers, each initialized to `null`. The registers dedicated for depositing allow to keep track of indices of used registers, and to obtain next registers still available for deposits by the operation of updating lists. It is an open problem if there exist algorithmic solutions to Mining-Names in an asynchronous distributed system with finitely many shared read-write registers and processes prone to crashes.

References

[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 *Proceedings of the* 18*th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 91–103, 1999.

[3] Yehuda Afek, Pazi Boxer, and Dan Touitou. Bounds on the shared memory requirements for long-lived adaptive objects (extended abstract). In *Proceedings of the* 19*th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 81–89, 2000.

[4] Yehuda Afek and Yaron De Levie. Efficient adaptive collect algorithms. *Distributed Computing*, 20(3):221–238, 2007.

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

[6] Yehuda Afek, Gideon Stupp, and Dan Touitou. Long-lived adaptive collect with applications. In *Proceedings of the* 40*th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 262–272, 1999.

[7] Yehuda Afek, Gideon Stupp, and Dan Touitou. Long lived adaptive splitter and applications. *Distributed Computing*, 15(2):67–86, 2002.

[8] Marcos Kawazoe Aguilera. A pleasant stroll through the land of infinitely many creatures. *SIGACT News*, 35(2):36–59, 2004.

[9] Dan Alistarh. The renaming problem: Recent developments and open questions. *Bulletin of EATCS*, 117:102–141, 2015.

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

[11] 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* 30*th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 239–248, 2011.

[12] Dan Alistarh, James Aspnes, George Giakkoupis, and Philipp Woelfel. Randomized loose renaming in $O(\log \log n)$ time. In *Proceedings of the* 32*nd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 200–209, 2013.

[13] Dan Alistarh, Hagit Attiya, Seth Gilbert, Andrei Giurgiu, and Rachid Guerraoui. Fast randomized test-and-set and renaming. In *Proceedings of the 24th International Symposium on Distributed Computing (DISC)*, volume 6343 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2010.

[14] James Aspnes, Gauri Shah, and Jatin Shah. Wait-free consensus with infinite arrivals. In *Proceedings on the 34th ACM Symposium on Theory of Computing (STOC)*, pages 524–533, 2002.

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

[16] Hagit Attiya, Armando Castañeda, Maurice Herlihy, and Ami Paz. Bounds on the step and namespace complexity of renaming. *SIAM Journal on Computing*, 48(1):1–32, 2019.

[17] Hagit Attiya, Faith Ellen Fich, and Yaniv Kaplan. Lower bounds for adaptive collect and related objects. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 60–69, 2004.

[18] Hagit Attiya and Arie Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM Journal on Computing*, 31(2):642–664, 2001.

[19] Hagit Attiya and Arie Fouren. Algorithms adapting to point contention. *Journal of the ACM*, 50(4):444–468, 2003.

[20] Hagit Attiya, Arie Fouren, and Eli Gafni. An adaptive collect algorithm with applications. *Distributed Computing*, 15(2):87–96, 2002.

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

[22] Hagit Attiya and Ami Paz. Counting-based impossibility proofs for set agreement and renaming. *Journal of Parallel and Distributed Computing*, 87:1–12, 2016.

[23] Hagit Attiya and Sergio Rajsbaum. The combinatorial structure of wait-free solvable tasks. *SIAM Journal on Computing*, 31(4):1286–1313, 2002.

[24] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. John Wiley, 2nd edition, 2004.

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

[26] Alex Brodsky, Faith Ellen, and Philipp Woelfel. Fully-adaptive algorithms for long-lived renaming. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, volume 4167 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2006.

[27] Harry Buhrman, Juan A. Garay, Jaap-Henk Hoepman, and Mark Moir. Long-lived renaming made fast. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 194–203, 1995.

[28] James E. Burns and Gary L. Peterson. The ambiguity of choosing. In *Proceedings of the* 8*th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 145–157, 1989.

[29] Michael R. Capalbo, Omer Reingold, Salil P. Vadhan, and Avi Wigderson. Randomness conductors and constant-degree lossless expanders. In *Proceedings of the* 34*th ACM Symposium on Theory of Computing (STOC)*, pages 659–668, 2002.

[30] Armando Castañeda, Maurice Herlihy, and Sergio Rajsbaum. An equivariance theorem with applications to renaming. *Algorithmica*, 70(2):171–194, 2014.

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

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

[33] Bogdan S. Chlebus and Dariusz R. Kowalski. Asynchronous exclusive selection. In *Proceedings of the* 27*th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 375–384, 2008.

[34] Bogdan S. Chlebus, Dariusz R. Kowalski, and Alexander A. Shvartsman. Collective asynchronous reading with polylogarithmic worst-case overhead. In *Proceedings of the* 36*th ACM Symposium on Theory of Computing (STOC)*, pages 321–330, 2004.

[35] Gregory V. Chockler and Dahlia Malkhi. Active Disk Paxos with infinitely many processes. *Distributed Computing*, 18(1):73–84, 2005.

[36] Wayne Eberly, Lisa Higham, and Jolanta Warpechowska-Gruca. Long-lived, fast, waitfree renaming with optimal name space and high throughput. In *Proceedings of the* 12*th International Symposium on Distributed Computing (DISC)*, volume 1499 of *Lecture Notes in Computer Science*, pages 149–160. Springer, 1998.

[37] Eli Gafni, Michael Merritt, and Gadi Taubenfeld. The concurrency hierarchy, and algorithms for unbounded concurrency. In *Proceedings of the* 20*th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 161–169, 2001.

[38] Maryam Helmi, Lisa Higham, and Philipp Woelfel. Space bounds for adaptive renaming. In *Proceedings of the* 28*th International Symposium on Distributed Computing (DISC)*, volume 8784 of *Lecture Notes in Computer Science*, pages 303–317. Springer, 2014.

[39] Maurice Herlihy, Dmitry Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann, 2013.

[40] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, 1999.

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

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

[43] Michael Merritt and Gadi Taubenfeld. Resilient consensus for infinitely many processes. In *Proceedings of the* 17*th International Conference on Distributed Computing (DISC)*, volume 2848 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2003.

[44] Michael Merritt and Gadi Taubenfeld. Computing with infinitely many processes. *Information and Computation*, 233:12–31, 2013.

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

[46] Mark Moir and Juan A. Garay. Fast, long-lived renaming improved and simplified. In *Proceeding of the* 10*th International Workshop on Distributed Algorithms (WDAG)*, volume 1151 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 1996.

[47] Michael E. Saks, Nir Shavit, and Heather Woll. Optimal time randomized consensus - making resilient algorithms fast in practice. In *Proceedings of the* 2*nd ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 351–362, 1991.