Long-Lived Renaming Made Fast

Harry Buhrman* Juan A. Garay† Jaap-Henk Hoepman* Mark Moir‡

Abstract

In the long-lived renaming problem — a generalization of the classical one-time renaming problem — n processors with unique names ranging over a *source name space* $\{0, \ldots, S-1\}$ repeatedly acquire and release unique names from a (smaller) *destination name space* $\{0, \ldots, D-1\}$. It is assumed that at most k out of n processors concurrently request or hold names. An efficient renaming protocol provides a useful front-end for protocols whose time complexity depends on the size of the name space containing the participating processes.

We consider long-lived renaming in the context of asynchronous, shared-memory multiprocessing systems that provide only read and write operations. A renaming protocol is *fast* iff the time complexity of acquiring and releasing a name is polynomial in k and independent of n and S. We present a wait-free, read/write protocol for long-lived renaming that achieves a destination name space of size $O(k^2)$ with time complexity $O(k^3)$. If S is polynomial in k, we further improve the time-complexity to $O(k \log k)$. This shows, for the first time, that fast, read/write protocols for long-lived renaming exist. Part of our wait-free solution uses mutual exclusion tournament trees, where we apply hashing based on polynomials over finite fields to avoid blocking. This technique may be of general interest.

1 Introduction

In the *one-time renaming problem* [ABND+90, BND89, BG93, PPTV94], n processes with unique identifiers in the range $\{0, \ldots, S-1\}$ obtain distinct names from the smaller range $\{0, \ldots, D-1\}$ at most once. The *long-lived renaming problem* generalizes the one-time renaming problem by allowing processes to repeatedly acquire and release names. It is assumed that at most k out of n processes acquire or hold names concurrently.

An efficient renaming protocol is a useful front-end for protocols whose time complexity depends on the size of the name space containing the participating processes. In particular, Moir and Anderson have shown that the overhead associated with accessing a resilient shared object can be reduced by combining a long-lived renaming protocol with a shared object implementation for fewer processes [AM94]. Efficient renaming protocols can also be useful, for example, in Unix-based multiprocessing systems. In such systems, processes have unique identifiers from a large range, but the number of processes that run concurrently is much smaller. Thus, if a group of processes participate in a computation whose time complexity is dependent on the size of the name space containing those processes, then using a renaming protocol to reduce the size of that name space can dramatically improve performance.

We consider long-lived renaming protocols for asynchronous, shared-memory, multiprocessing systems. The long-lived renaming problem has been considered in this context by Moir and Anderson [MA94], and in message-passing systems by Bar-Noy *et al.* [BNDKP91]. We call a renaming protocol *fast* iff acquiring and releasing a name takes time $O(p(k))$ where p is a polynomial that is independent of both n, the total number of processes, and S, the size of their original name space. For systems supporting primitives such as *Test&Set*, Moir and Anderson present renaming protocols that are both fast and long-lived. However, protocols that employ such strong operations are not as widely applicable or as portable as protocols that employ only reads and writes. We are therefore motivated to study fast, long-lived renaming protocols that use only read and write operations. Moir and Anderson's only read/write, long-lived renaming protocol, hereafter called MA, is not fast: it yields a name space of size $k(k+1)/2$ with time complexity $O(kS)$.

In this paper, we present two long-lived renaming protocols: SPLIT and FILTER. These two protocols, combined with the non-fast protocol MA [MA94], yield the first fast *and* long-lived renaming protocol that is based on reads and writes. Our protocol renames k processes

*CWI, Amsterdam, The Netherlands. Partially supported by the Dutch foundation for scientific research (NWO) through NFI Project ALADDIN, under contract number NF 62-376. Email: {buhrman,jhh}@cwi.nl.

†IBM T. J. Watson Research Center, Yorktown Heights, USA. Email: garay@watson.ibm.com.

‡The University of North Carolina at Chapel Hill, Chapel Hill, North Carolina, USA. Work supported, in part, by NSF Contract CCR 9216421. Email: moir@cs.unc.edu.

to a name space of size $k(k+1)/2$, with time complexity $O(k^3)$. If the size S of the original name space is polynomial in k, protocol FILTER on its own renames k processes to a name space of size $O(k^2)$ with time-complexity $O(k \log k)$. We now present a brief description of the SPLIT and FILTER protocols, before presenting them in detail.

The SPLIT protocol uses a collection of building blocks called "splitters". Each splitter, if accessed by at most ℓ processes concurrently, dynamically partitions these processes into three output sets, each of which contains at most $\ell - 1$ processes. SPLIT employs a k-deep tree of splitters. Each leaf of the tree corresponds to a single name. A process p acquires a name by traversing the tree from the root to a leaf, accessing one splitter at each level. The splitter accessed at each level below the root is determined by the output set assigned by the splitter accessed at the previous level. After proceeding through $k-1$ levels of the tree, a process is guaranteed to be in an output set that contains no other processes. Accessing a single splitter takes constant time. Thus, SPLIT renames to a name space of size 3^{k-1} with time complexity $O(k)$. A process releases a name by releasing each of the splitters it accessed. This also takes $O(k)$ time, so the SPLIT renaming protocol is fast.

FILTER is based on a set of mutual exclusion tournament trees — one for each destination name. In order to acquire a name, a process p competes for each of a set N_p of names by participating in the mutual exclusion tree associated with each name in N_p. We present a modified version of Peterson and Fischer's mutual exclusion tournament trees [PF77] that enables a process p to compete for all of the names in N_p "in parallel". N_p is chosen in such a way that, while process p is attempting to acquire a name, there is always an $x \in N_p$ for which no other process contends concurrently. This is achieved by the use of a special hashing technique that is based on unique polynomials over a finite field [EFF85]. Because there is always some tree T in which p is participating alone, the FIFO property of the mutual exclusion ensures that next time p participates in tree T, p makes progress towards the critical section of T. Therefore, p can eventually acquire a name.

Usually the size S of the source name space is bounded from above by a function f of k. If $f \in O(k^c)$, then FILTER renames to a name space of size $O(c^2k^2)$ in time $O(k \log k)$. If $f = 3^{k-1}$, then FILTER renames to a name space of size $2k^4$ in time $O(k^3)$. If $f = 2k^4$, FILTER renames to a name space of size $72k^2$ in time $O(k \log k)$. Using protocol MA [MA94] we can further reduce the size of the name space to $k(k+1)/2$ in time $O(k^3)$. In practice we can assume $f \in O(3^k)$, so the first stage implemented by SPLIT is unnecessary.

The remainder of the paper is organized as follows. Section 2 contains definitions used in the paper. In Sections 3 and 4, we present the SPLIT and FILTER protocols, respectively. Concluding remarks appear in Section 5.

2 Definitions

We consider an asynchronous, shared memory, multiprocessing environment in which n processes communicate through shared variables that can be atomically read or written by any process. Each process p has a unique identifier from the set $\{0, \ldots, S-1\}$, where $S \geq n$.

An *operation pair* (E, R) on a shared object OBJ consists of two operations E (for *Enter*) and R (for *Release*). A process p that accesses an object OBJ via an operation pair (E,R) is required to alternately execute E and R on OBJ; p's first operation on OBJ must be $E(OBJ)$. The following predicates describe p's position in its access cycle for object OBJ.

Inside(OBJ, p)**:** p has completed $E(OBJ)$, but has not yet started $R(OBJ)$, and

Using(OBJ, p)**:** p is executing $E(OBJ)$ or $R(OBJ)$, or *Inside*(OBJ, p) holds.

Initially, $\neg$*Using*(OBJ, p) holds.

A solution to the long-lived renaming problem consists of a wait-free implementation of the operation pair (*GetName*(p),*ReleaseName*(p)) on a renaming object RN that is shared by n processes. In order to obtain a name from $\{0, \ldots, D-1\}$, process p executes *GetName*(p) and assigns the return value to $name_p$. A process releases the name it holds by executing *ReleaseName*, thus freeing this name for later use by another process. Note that (*GetName*,*ReleaseName*) is an operation pair, so a process is required to release a name before it tries to acquire another name.

It is assumed that at most k processes concurrently access RN. The implementation of *GetName* and *ReleaseName* is required to ensure that distinct processes p and q do not hold the same name concurrently. That is, the following assertion is required to be an invariant:

$$p \neq q \text{ and } Inside(RN, p) \text{ and } Inside(RN, q) \text{ implies } name_p \neq name_q.$$

We measure the time complexity of our implementations by giving an upper bound on the number of shared memory accesses performed by any operation execution. We call a renaming protocol *fast* if both *GetName* and *ReleaseName* have time-complexity polynomial in k and independent of S, the size of the original name space, and n, the total number of processes.

In our protocols, we use $\leftarrow$ to denote assignment, *name* to denote local variables, NAME to denote shared variables, and **name** for keywords. Each labelled statement in our figures is assumed to be executed atomically. Note that each such statement contains at most one access of a shared variable.

For strings s of finite length over a finite alphabet, we use $|s|$ to denote their length, $(s\,b)$ to denote the result of appending symbol b to string s, $s[1{:}k]$ to denote the string consisting of the first k symbols of s, and $s[i]$ to denote the ith symbol of s (for example, $s[1]$ is the first symbol of s). Also, we use A^h to denote the set of h-length strings over alphabet A and $A^{\leq h}$ to denote the set of such strings of length at most h. Finally, $(\#x : x \in X :: P(x))$ denotes the number of $x \in X$ such that $P(x)$ holds, and $\|X\|$ denotes the cardinality of X.

3 Renaming to 3^{k-1} Names

In this section, we present the SPLIT protocol, which, for any S, renames to $3^k - 1$ names with time complexity $O(k)$. SPLIT uses a "building block" similar to the one employed by Moir and Anderson [MA94]. Processes accessing our building block, which is presented in Section 3.1 below, are dynamically partitioned into three output sets, denoted -1, 0 and 1. The building block ensures that, if at most ℓ processes access a building block concurrently, then each output set contains at most $\ell - 1$ processes at any time. A process leaves a group by "releasing" the building block. We now define the building block formally, and show how it is used to implement long-lived renaming.

An implementation of building block B consists of paired operations $Enter(B,p)$, returning -1, 0, or 1, and $Release(B,p)$. In defining the correctness condition below for building block B, we use $e_p(B)$ to denote the value returned by the most recent call to $Enter(B,p)$. The implementation of B is required to ensure that, if at most ℓ processes concurrently access B, then the following assertion is an invariant for all $d \in \{-1,0,1\}$:

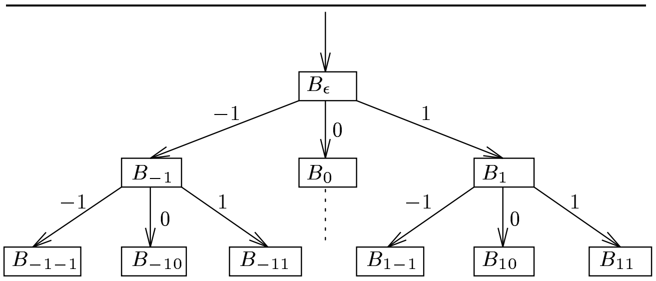
$$(\#p :: Inside(B,p) \ \wedge\ e_p(B) = d) < \ell\ .$$

Protocol SPLIT, shown in Figure 1, uses a tree of depth $k-1$ of building-blocks to rename k processes. Each interior node has three children — one for each of the output sets -1, 0, and 1. Each node in the tree is labelled by a string from $\{-1,0,1\}^{\leq k-1}$. The building block at the root is labelled as B_ϵ. Each other building block B is labelled as $(s\,d)$, where s is the label of B's parent, and d is -1, 0, or 1, depending on which output set B is associated with. Thus, the leaves of the tree are labelled by strings of length $k-1$ over the alphabet $\{-1,0,1\}$. To simplify reasoning later, we consider such a leaf to be vacuous building block, used by the processes reaching it.

In order to acquire a name, process p starts at the root of the tree, and then traverses the tree until it reaches a leaf. At each level h, p chooses the child in level $h+1$ associated with the output set returned by the building block accessed at level h. The label s of the leaf reached is used to compute the name to be returned as follows:

$$\overline{s} = \sum_{i=1}^{k-1} (1 + s[i]) 3^{i-1}\ .$$

As the depth of the tree is $k-1$, the properties of the building block guarantee that no other process is currently holding the same name. To release a name, a process simply releases all building blocks it accessed in acquiring that name.



Local variables (static):
  $s \in \{-1,0,1\}^{\leq k-1}$;
  $e \in \{-1,0,1\}$;
  $j \in \{1,\ldots,k\}$;

GetName(p) :
  $s \leftarrow \epsilon$;
  for $j \leftarrow 1$ **to** $k-1$
  do $e \leftarrow Enter(B_s, p)$;
    $s \leftarrow (s\,e)$;   (* Append e to s *)
  return $\overline{s}$;   (* Compute name from path *)

ReleaseName(p) :
  for $j \leftarrow k-1$ **downto** 1
  do $Release(B_s, p)$;
    $s \leftarrow s[1{:}|s|-1]$;   (* Discard last label in s *)

Figure 1: Renaming protocol SPLIT for process p

Lemma 1 *For all $s \in \{-1,0,1\}^{\leq k-1}$, the following assertion is an invariant.*

$$(\#p :: Using(B_s,p)) \leq k - |s|.$$

Proof: By induction on $|s|$. For $|s| = 0$ (i.e. $s = \epsilon$), the lemma holds by assumption that at most k processes concurrently hold or attempt to acquire names. We inductively assume that the lemma holds for all strings over $\{-1,0,1\}$ of length at most m. Let s be any string over $\{-1,0,1\}$ of length $m+1$, and set $t = s[1{:}m]$. By the inductive hypothesis, the following assertion is an invariant.

$$(\#p :: Using(B_t,p)) \leq k - |t|$$

We assume that the building block B_t is correct. Therefore, the correctness condition for the building block implies that the following property is an invariant.

$$(\#p :: \mathit{Inside}(B_t, p) \wedge e_p(B_t) = s[m+1]) \leq k - |t| - 1.$$

It is easy to show that $\mathit{Using}(B_s, p)$ implies $\mathit{Inside}(B_t, p) \wedge e_p(B_t) = s[m+1]$. Therefore

$$(\#p :: \mathit{Using}(B_s, p)) \leq k - |t| - 1 = k - |s|$$

is an invariant. ■

In the next section, we present a wait-free implementation of building block B. This implementation performs $O(1)$ shared accesses per operation, which allows us to prove the following theorem.

Theorem 2 *Protocol* SPLIT *implements wait-free, long-lived renaming to* 3^{k-1} *names in time* $O(k)$*, and hence is fast.*

Proof: Given that each building block B can be implemented in a wait-free manner, it is easy to see that SPLIT is wait-free. As

$$\overline{s} = \sum_{i=1}^{k-1} (1 + s[i]) 3^{i-1} < 3^{k-1},$$

SPLIT yields a name space of size 3^{k-1}. Also, it is easy to show that $s \neq t$ implies $\overline{s} \neq \overline{t}$. Thus, if p has name $\overline{s}$, then p is in building block B_s. Therefore, Lemma 1 implies that distinct processes do not concurrently hold the same name. The building block implementation presented in Section 3.1 performs at most 9 shared variable accesses per operation. Each operation of SPLIT accesses $k-1$ building blocks. Thus, SPLIT is a fast, long-lived renaming protocol. ■

3.1 Implementing the "Splitter"

In this section, we present a building block B, which partitions processes into three sets, -1, 0, and 1. In order to join a set, process p calls $\mathit{Enter}(B, p)$ and joins the output set associated with the value returned. Later, p leaves that set by calling $\mathit{Release}(B, p)$. As stated earlier, the implementation is required to ensure that, provided at most ℓ processes concurrently access the building block, no output set contains more than $\ell - 1$ processes at any time. The building block implementation appears in Figure 2.

In accessing the building block, processes attempt to pass "advice" to each other about which output set can be safely joined. For example, if a process p chooses set 1, then p can safely advise another process to join set -1 because all processes are not in set -1. Similarly, if p leaves set 1, then p can advise another process to join set 1. Because of the difficulty of correctly passing advice between asynchronous processes using only read and write operations, the building block employs an "interference detection" mechanism that allows the advice to be incorrect in all but one special case. In this special case, described below, the processes execute

```
Shared multi-writer variables:
      LAST ∈ {0, ..., S − 1} ;
      ADVICE[1] ∈ {⊥, −1, 1} ; initially 1
      ADVICE[2] ∈ {−1, 1} ; initially 1
Local variables (static):
      advice ∈ {−1, 1} ;
      adv2 ∈ {true, false} ;

Enter(B, p) :
1:    LAST ← p ;
2:    advice ← ADVICE[1] ;
3:    if advice = ⊥ then advice ← ADVICE[2] ;
4:    ADVICE[1] ← −advice ;
5:    adv2 ← (LAST = p) ;
6:    if adv2 then ADVICE[2] ← −advice ;
7:    if LAST = p
          then return advice ;
          else return 0 ;

8:        (* Working Section *)

Release(B, p) :
9:    if LAST = p
10:       then ADVICE[1] ← advice ;
11:   if ¬adv2 then ADVICE[1] ← ⊥ ;
12:       (* Remainder Section *)
```

Figure 2: Process p's code for building block B

Enter "sequentially" (the steps of one process are not interleaved with those of another). This lack of interleaving allows correct advice to be passed between the processes in the special case, thereby ensuring that each output set contains at most $\ell - 1$ processes at any time. Below, we describe the building block implementation in more detail, before giving a correctness proof.

LAST stores the identifier of the last process to enter the building block. Process p writes its identifier to LAST upon entry, and reads it again at line 7 before deciding on a return value. If p reads LAST $\neq p$ (i.e., p detects "interference" from another process), then p returns 0, otherwise p returns *advice*. It is easy to show that at most $\ell - 1$ processes are in output set 0 at any time. To see why this is so, observe that if ℓ processes are inside the building block, then the last process q to assign LAST reads LAST $= q$ (does not detect any interference). In this case, q returns *advice*, which is easily shown to be non-zero. Thus, if ℓ processes are inside the building block, then at least one of the processes is not in output set 0.

To see that at most $\ell - 1$ processes are in output set 1 at any time is more complicated (the case for set -1 is symmetric). First, note that if ℓ processes are in output set 1, then each process p read LAST $= p$ at line p.7. This is only possible if ℓ processes execute *Enter* "sequentially" — that is, each process executes step 7 before the following process executes step 1. Consider the second-to-last process q to execute *Enter*. Because q detects no interference, and because q joins group 1, it is easy to show that q assigns -1 to $\mathsf{ADVICE}[1]$ and $\mathsf{ADVICE}[2]$ at steps 4 and 6 respectively. Informally, this represents advice to the last process p to join set -1. If this advice remains until p executes *Enter*, then it is easy to see that p joins output set -1, contradicting the assumption that all ℓ processes are in set 1 concurrently. A key property in the correctness proof presented below, is that when p executes *Enter* in this scenario, q's advice is intact. This property is based on the observation that, apart from the $\ell-1$ processes already in an output set after q executes step 7, at most one process accesses the building block concurrently before p executes 1. Because none of the remaining $\ell-1$ processes take any steps in this interval, it is straightforward to show that $\mathsf{ADVICE}[1] = -1$ or $\mathsf{ADVICE}[1] = \perp \ \wedge\ \mathsf{ADVICE}[2] = -1$ holds when p begins executing *Enter*. In either case, p joins set -1, and therefore does not violate the correctness condition for the building block. We now present a formal correctness proof for the building block.

3.2 Correctness of the Building Block

In this section, we assume that $(\#p :: \mathit{Using}(B,p)) \leq \ell$ holds throughout a fixed, but arbitrary execution (modelled by $\Rightarrow$) of the building block implementation shown in Figure 2. To show that the implementation is correct, we prove that, for each $d \in \{-1,0,1\}$, the following assertion is an invariant.

$$(\#p :: \mathit{Inside}(B,p) \ \wedge\ e_p(B) = d) \leq \ell - 1\ .$$

In the following, we use p to denote both process p and an invocation of $\mathit{Enter}(B,p)$ and the corresponding $\mathit{Release}(B,p)$. In the rare instances where we consider different invocations by the same process, we distinguish the invocations (e.g., using p and p'). We use $e_p(B)$ to denote the value returned by invocation p of $\mathit{Enter}(B,p)$; $p.\ell$ denotes the atomic action of executing line ℓ by invocation p; and $p@\ell$ means that the next line to be executed by process p is ℓ. For a specific invocation, we use $\mathit{Using}(B,p)$ (respectively, $\mathit{Inside}(B,p)$) to indicate that p is using (resp., inside) B during that invocation. Finally, advice_p denotes the value of local variable *advice* held by processor p during this particular invocation.

The following claim is used to prove the above property for $d = 0$.

Claim 3 *For an arbitrary invocation p, $e_p(B) = 0$ iff there exists an invocation q such that $p.1 \Rightarrow q.1 \Rightarrow p.7$.*

Proof: If there is a q such that $p.1 \Rightarrow q.1 \Rightarrow p.7$, then at line 7, p reads $\mathsf{LAST} \neq p$ and $e_p(B) = 0$, otherwise p reads $\mathsf{LAST} = p$ and $e_p(B) \neq 0$. ■

For the case where $d = -1$ or $d = 1$ we first prove the following lemma.

Lemma 4 *Let $d \neq 0$. Suppose that at some point there are $\ell-1$ invocations $p_1, \ldots, p_{\ell-1}$ with $p_i@\{8,\ldots,10\}$ and $e_{p_i}(B) = d$, and there is an invocation q with $q@4$ and $p_i.7 \Rightarrow q.1$ for $1 \leq i \leq \ell-1$. Then $\mathit{advice}_q = -d$*

Proof: Without loss of generality, let $p_i.1 \Rightarrow p_{i+1}.1$ for $1 \leq i \leq \ell-2$. By Claim 3 and $e_{p_i}(B) \neq 0$ for $1 \leq i \leq \ell-1$ we also have $p_i.7 \Rightarrow p_{i+1}.1$ for $1 \leq i \leq \ell-2$. In other words, $p_1, \ldots, p_{\ell-1}$ must have entered sequentially. Towards a contradiction, suppose that at time t for the first time the conditions of the lemma hold, but $\mathit{advice}_q = d$.

There are four cases for the last write to $\mathsf{ADVICE}[1]$ before q reads it. Either there is some invocation r writing $\mathsf{ADVICE}[1]$ after $p_{\ell-1}$ did (case 1, 2, and 3 below), or not (case 4 below). If some invocation r writes $\mathsf{ADVICE}[1]$ after $p_{\ell-1}$ but before q reads it, it writes $\mathsf{ADVICE}[1]$ in line 10 (case 1), in line 11 (case 2), or in line 4 (case 3). In the first three cases, by $(\#p :: \mathit{Using}(B,p)) \leq \ell$, r must finish before q starts. Hence if r executes line 10 or 11 then $r.10, r.11 \Rightarrow q.1$. Also in these three cases, as $e_{p_{\ell-1}}(B) \neq 0$, by Claim 3 we have either $r.1 \Rightarrow p_{\ell-1}.1$ or $p_{\ell-1}.7 \Rightarrow r.1$.

1. For some r, $p_{\ell-1}.4 \Rightarrow r.10 \Rightarrow q.2$. Then q reads $\mathsf{ADVICE}[1] = \mathit{advice}_r = d$. If at line 9, $\mathsf{LAST} = r$ then also at line 7 and 5. Hence $e_r(B) = d$. But then at a time $t' < t$ we have an earlier bad case contrary to assumption. For if $p_{\ell-1}.7 \Rightarrow r.1$ then at time t' where $r@4$ we have $\mathit{advice}_r = d$ and still $p_i@\{8,\ldots,10\}$ and $e_{p_i}(B) = d$ for $i = 1,\ldots,\ell-1$. And if $r.1 \Rightarrow p_{\ell-1}.1$ (the only other case, see above) as $\mathsf{LAST} = r$ at line 9 we must have $r.9 \Rightarrow p_{\ell-1}.1$. But then at time t' where $p_{\ell-1}@4$ we have $\mathit{advice}_{p_{\ell-1}} = d$ by assumption that $e_{p_{\ell-1}}(B) = d$ and still $p_i@\{8,\ldots,10\}$ and $e_{p_i}(B) = d$ for $i = 1,\ldots,\ell-2$ and (because $p_{\ell-1}.4 \Rightarrow r.10$) also $r@\{8,\ldots,10\}$ and $e_r(B) = d$.
2. For some r, $p_{\ell-1}.4 \Rightarrow r.11 \Rightarrow q.2$. Then q reads $\mathsf{ADVICE}[1] = \perp$ and hence reads $\mathsf{ADVICE}[2] = d$ at line 3. Then this value is written there by $p_{\ell-1}$. But then $e_{p_{\ell-1}}(B) = -d$, a contradiction. To see why $p_{\ell-1}$ is the last to write $\mathsf{ADVICE}[2]$, let us suppose to the contrary that there is a r' with $p_{\ell-1}.6 \Rightarrow r'.6 \Rightarrow q.3$. Because r executes line 11, it does not execute line 6, so $r' \neq r$. As by assumption r is the last to write $\mathsf{ADVICE}[1]$ at line 11, $r'.2 \Rightarrow r.11$. By $(\#p :: \mathit{Using}(B,p)) \leq \ell$ then also $r'.6 \Rightarrow r.1$, and using $p_{\ell-1}.6 \Rightarrow r'.6$ we get $p_{\ell-1}.1 \Rightarrow r.1$. Again by $(\#p :: \mathit{Using}(B,p)) \leq \ell$ then r reads $\mathsf{LAST} = r$ at line 5 thus setting $\mathit{adv2}_r = \mathit{true}$, contrary to the assumption that r executes step 11.
3. For some $r \neq p_{\ell-1}$, $p_{\ell-1}.4 \Rightarrow r.4 \Rightarrow q.2$. As we have $(\#p :: \mathit{Using}(B,p)) \leq \ell$, if $p_{\ell-1}.7 \Rightarrow r.1$ then r reads $\mathsf{LAST} = r$ at line 9 and execute line 10. But then $r.10 \Rightarrow q.2$; this is case 1 above. Now consider $r.1 \Rightarrow p_{\ell-1}.1$ (the only other case, see above). By

$p_{\ell-1}.4 \Rightarrow r.4$ we get $r.1 \Rightarrow p_{\ell-1}.1 \Rightarrow r.5$, hence r reads $\mathsf{LAST} \neq r$ at line 5. Then $adv2_r = false$ and r executes line 11. But then $r.11 \Rightarrow q.2$; this is case 2 above.

4. q reads from $\mathsf{ADVICE}[1]$ what $p_{\ell-1}$ writes there (i.e., none of the above cases) at $p_{\ell-1}.4$ (because $p_{\ell-1}@\{8,\ldots,10\}$, $p_{\ell-1}$ did not execute line 10 yet). As $e_{p_{\ell-1}}(B) = d$ this must be $-d$, a contradiction.

This completes the proof of Lemma 4. ■

Theorem 5 *For each $d \in \{-1,0,1\}$, the following assertion is invariant.*

$$(\#p :: Inside(B,p) \ \wedge \ e_p(B) = d) \leq \ell - 1.$$

Proof: Suppose for $d = 0$ the theorem does not hold. Then $(\#p :: Inside(B,p) \ \wedge \ e_p(B) = 0) = \ell$ at some point. Of all these invocations p, let q be the last to write LAST in step 1 (i.e., $p \neq q$ implies $p.1 \Rightarrow q.1$). Then by $(\#p :: Using(B,p)) \leq \ell$ there is no invocation r such that $q.1 \Rightarrow r.1 \Rightarrow q.7$. Hence by Claim 3, $e_q(B) \neq 0$, a contradiction.

Towards a contradiction in the case where $d = -1$ or $d = 1$, suppose $(\#p :: Inside(B,p) \ \wedge \ e_p(B) = d) = \ell$ at some point. Then similar to the proof of Lemma 4, by Claim 3 $p_1,\ldots,p_\ell$ must have entered sequentially, i.e. $p_i.7 \Rightarrow p_{i+1}.1$ for $1 \leq i \leq \ell - 1$. Then at $p_\ell@4$ we have $p_i@\{8,\ldots,10\}$ and $e_{p_i}(B) = d$ for $1 \leq i \leq \ell - 1$ and hence by Lemma 4 $advice_{p_\ell} = -d$. This contradicts $e_{p_\ell}(B) = d$. ■

4 Reducing the Name Space

In this section, we present the long-lived renaming protocol FILTER. An instance of the FILTER protocol is specified by two parameters d and z. Given k and S, any choice of d and z satisfying several requirements (described in Section 4.1 below) yields a long-lived renaming algorithm. In particular, if $S \leq 3^k - 1$ (which can be achieved using SPLIT), then d and z can be chosen so that FILTER renames to a name space of size $2k^4$ with time complexity $O(k^3)$. Also, for $S = 2k^4$, d and z can be chosen so that FILTER renames to a name space of size $72k^2$ with time complexity $O(k \log k)$. In Section 4.4 we explain how these instances can be used to achieve renaming to $k(k+1)/2$ names with $O(k^3)$ time complexity. We now give a brief overview of the FILTER protocol, before presenting it in detail.

FILTER uses a collection of mutual exclusion tournament trees — one tree T_m for each name m in $\{0,\ldots,D-1\}$ (recall that D is the size of the destination name space). In order to acquire a name m using the FILTER protocol, a process p competes in the mutual exclusion tree T_m associated with that name.

The use of mutual exclusion in a wait-free protocol might seem counterintuitive. However, as described in detail below, each process p competes "in parallel" for each of a set N_p of names. This is achieved by the use of a modified version of Peterson and Fischer's [PF77] mutual exclusion trees. The modification allows a process to detect that it is blocked in one tree, and to attempt to acquire another name from N_p by continuing to compete in another tree associated with that name. The collection of sets N_p is constructed using a special hashing technique involving polynomials over a finite field, such that no set is covered by the union of $k-1$ others. Collections with that property were studied by Erdös *et al.* [EFF85, BLS93]. In our application this property ensures that, at any time, there is some name $m \in N_p$ for which p is competing alone.

Section 4.1 below explains this hashing technique in detail. Then, in Section 4.2, we present the modified mutual exclusion tree and the FILTER protocol. Correctness of the protocol is proven in Section 4.3. Finally, in Section 4.4, we discuss the performance of FILTER under different assumptions on the relationship between S and k.

4.1 Hashing Names to Sets of Names

The hashing technique used to assign a set of names to each process uses two parameters d and z, which are chosen based on particular values of k and S. As is discussed later, the choices of d and z for given values of k and S influence the time and space complexity of the resulting instance of the FILTER protocol, as well as the size of the destination name space.

We now show how the set of names N_p, for which process p competes, is defined, and state the constraints on the parameters d and z as we proceed. First, let $\mathsf{GF}(z)$ be a finite field, with z a prime (i.e., the elements of the field range over the set $\{0,\ldots,z-1\}$). Each process p is assigned a polynomial $Q_p(x) = a_d x^d + \cdots a_1 x + a_0$ of degree d over $\mathsf{GF}(z)$ ($0 \leq a_i < z$, and multiplication and addition are performed modulo z), such that the polynomials assigned to distinct processes differ in at least one coefficient. If

$$S \leq z^{d+1} \ , \qquad (1)$$

this can be achieved by assigning, for each process p, and for each i, $0 \leq i \leq d$, $a_i = (p \text{ div } z^i) \bmod z$.

We define $n_p(x) = z * x + Q_p(x)$ and define $N_p = \{n_p(0),\ldots,n_p(2d(k-1)-1)\}$. Because $Q_p(x) < z$ and $Q_q(y) < z$, it follows that $n_p(x) = n_q(y)$ iff $x = y$ and $Q_p(x) = Q_q(y)$. Thus, there are $2d(k-1)$ distinct names in N_p. Also, for distinct polynomials Q_p and Q_q of degree d over a finite field $\mathsf{GF}(z)$ with z prime, there are at most d values of x such that $Q_p(x) = Q_q(x)$ [Coh74]. Recall that for $p \neq q$, Q_p and Q_q are distinct. Then

$$z \geq 2d(k-1) \qquad (2)$$

implies $\|N_p \cap N_q\| \leq d$.

Furthermore, suppose P is an arbitrary set of $k-1$ processes such that $p \notin P$. Then there are at most $d(k-1)$ names $n_p(x) \in N_p$ which are also a member of some N_q with $q \in P$. As $\|N_p\| = 2d(k-1)$, this implies that there exist at least $d(k-1)$ names $n_p(i) \in N_p$ which are not a member of any N_q with $q \in P$.

In the FILTER protocol presented in the next section, process p competes only for names in N_p. Because at any time while p is attempting to acquire a name, at most $k-1$ other processes acquire or hold names, the property above implies that at any time, there are $d(k-1)$ names in N_p for which no other process is competing. This property is crucial in proving that process p can always acquire a name. Note that if we had required $z > d(k-1)$ a similar argument would have shown that there is at least one name in N_p for which no other process is competing. However, taking $z \geq 2d(k-1)$ allows us to arrive at a better bound on the time complexity in Theorem 10, at the expense of a small increase in size of the resulting name space.

The largest name competed for by any p is the maximum of $n_p(x) = z{*}x + Q_p(x)$ over all p and all x ranging over $0 \leq x \leq 2d(k-1)-1$. Clearly $Q_p(x)$ is bounded by z, so this shows that no process competes for a name larger than $z2d(k-1)$. Thus, an instance of the FILTER protocol specified by d and z achieves a destination name space of size

$$D = z2d(k-1)\ . \tag{3}$$

To achieve the smallest destination name space possible, d and z should be chosen to minimize $z2d(k-1)$, while satisfying (1) and (2) and the requirement that z be prime. In Section 4.4, we discuss various settings of d and z and the resulting name-space size for several combinations of k and S. We now present the FILTER protocol.

4.2 Protocol FILTER in Detail

For each name $m \in \{0, \ldots, D-1\}$, FILTER uses a binary mutual exclusion tournament tree T_m of $\lceil \log S \rceil$ levels. The leaves are at level 1 and the root is at level $\lceil \log S \rceil$. Each node in tree T_m is a distinct two-process mutual exclusion block ME with two "inputs" labelled 0 for left and 1 for right. Process names in $\{0, \ldots, S-1\}$ are mapped one-one to the $2^{\lceil \log S \rceil} \geq S$ "inputs" of the mutual exclusion blocks at level 1.

In order to compete in a tournament tree T_m, process p begins by entering the ME block connected to the input to which p's name maps. It passes to ME the direction β of this input. When p reaches the critical section of that ME block, p proceeds to the parent of the leaf and enters the ME block at that node, passing as a parameter β the direction from which it came. Process p continues up the levels of the tree until it reaches the critical section at the root. As shown in Lemma 6 below,

---

Enter(ME, β) :
  if $\mathsf{R}[1-\beta] = \mathit{nil}$ **then** $\mathsf{R}[\beta] \leftarrow \mathit{true}$
    else $\mathsf{R}[\beta] \leftarrow \beta \oplus \mathsf{R}[1-\beta]$;
  if $\mathsf{R}[1-\beta] \neq \mathit{nil}$ **then** $\mathsf{R}[\beta] \leftarrow \beta \oplus \mathsf{R}[1-\beta]$;

Check(ME, β) :
  return $(\mathsf{R}[1-\beta] = \mathit{nil}) \vee (\beta \oplus (\mathsf{R}[\beta] \neq \mathsf{R}[1-\beta]))$;

Release(ME, β) :
  $\mathsf{R}[\beta] \leftarrow \mathit{nil}$;

Figure 3: The 2-process mutual exclusion block ME.

---

p is guaranteed to be the only process in the critical section of the ME at the root of T_m at this point, so p can safely acquire name m.

To allow processes to compete in several mutual exclusion trees "in parallel", each node in the mutual exclusion trees contains a modified version of the two-process mutual exclusion algorithm of Peterson and Fischer [PF77]. Peterson and Fischer's algorithm is split into three procedures, *Enter*, *Check*, and *Release*, and uses multi-writer variables to avoid the costly search for an opponent. Except for these modifications, both algorithms are essentially the same. The three procedures are shown in Figure 3. In this figure, we use $\oplus$ to denote *exclusive or*, and $\vee$ to denote disjunction. We set $\mathit{true} = 1$ and $\mathit{false} = 0$.

Processes enter a mutual exclusion block ME either from the left (0) or from the right (1) subtree, as indicated by the parameter β. In order to compete in a particular two-process mutual exclusion block ME from direction β, process p calls *Enter*(ME, β) and then repeatedly calls *Check*(ME, β) until *Check* returns *true*. At this point p is in the critical section of ME. Later, p calls *Release*(ME, β) in order to release ME.

Because processes are mapped one-one to the inputs of the ME blocks at level 1, and because a process must reach the critical section of one ME block before accessing that block's parent, no two processes concurrently compete in any ME block from the same direction. Thus, each two-process ME block is accessed by at most two processes concurrently, one with $\beta = 0$ and the other with $\beta = 1$. This is the essence of the correctness proof, presented in Lemma 6 below, for each mutual exclusion tree.

Having described the tournament trees and the mutual exclusion blocks they use, we can now explain protocol FILTER. Process p competes "in parallel" in all the mutual exclusion trees associated with names in N_p (as defined in Section 4.1 above). This is achieved by proceeding through a tree as far as possible (until a call to *Check* returns *false*) and then switching to the tree for the next name. This is repeated until a name has been acquired (by reaching the critical section of the

```
repeat (* this starts a new round *)
    forall i ∈ {0, ..., 2d(k − 1) − 1} do
        if p did not enter tree n_p(i) yet
            then Enter tree n_p(i) at the appropriate leaf.
        Let ℓ be the last entered level of tree n_p(i).
        while ℓ ≠ ⌈log S⌉ and checking ME at level ℓ in
                tree n_p(i) returns true
            do Let ℓ ← ℓ + 1 and enter ME at level ℓ.
        if ℓ = ⌈log S⌉ and checking ME at level ℓ in
                tree n_p(i) returns true
            then Return name n_p(i)
until a name was found
```

Figure 4: FILTER protocol for process p.

tree for that name), or all names have been tried. In the latter case, process p then returns to the tree for p's first name and tries all the trees again. This is repeated until p acquires a name.

To release a name (not shown in Figure 4), process p releases all mutual exclusion blocks it entered when acquiring the name, including blocks in which p did not reach the critical section. Mutual exclusion blocks must be released in reverse of the order in which they were entered, i.e, the last block entered must be released first.

To see that p eventually acquires a name using protocol FILTER, recall that the set of trees competed for by p is chosen in such a way that, at any time, one of the trees in p's set is not being accessed by any other process (provided at most k processes request or hold names concurrently) . Suppose that at some point t, p is competing for name m, and no other process is competing for name m. If p fails to acquire a name, then p eventually tries to proceed in tree T_m. Because there was no other process competing for T_m at time t, the FIFO property of the 2-process mutual exclusion algorithms ensures that p is able to proceed at least one level in T_m. Repeating this argument, p eventually reaches the critical section of some tree, and therefore acquires a name. This argument is the essence of FILTER's correctness proof, which is presented in the next section.

4.3 Correctness Proof for FILTER

We say that a process p is *in tree T_m at level ℓ* if it has started entering ME at that level and has not yet started releasing that ME. Similarly, a process p is *in tree T_m at level* $\lceil \log S \rceil + 1$ if it has acquired and not yet released name m. We say that a process p is *in tree T_m* if it is in T_m at some level. We assume that there are at most k processes in all the trees at any time. A process p starts a new *round* whenever p attempts to advance in the tree for name $n_p(0)$. We say a process is *stuck at round r in tree T_m at level ℓ* if in two successive rounds $r-1$ and r, checking ME at level ℓ in tree T_m returns *false*. We call a process $p \in S$ a *visitor* of mutual exclusion block ME in tree T_m iff p enters T_m at one of the leaves of the subtree rooted at ME.

The following lemma states that T_m is indeed a tournament tree.

Lemma 6 *For all $m \in \{0, \ldots, D-1\}$ and all ℓ with $0 \le \ell \le \lceil \log S \rceil$ and all ME at level ℓ in tree T_m, no two different visitors $p, q \in \{0, \ldots, S-1\}$ of ME can be in T_m at level $\ell + 1$.*

Proof: Suppose the lemma does not hold for some tree T_m. Towards a contradiction pick the minimal ℓ for which two or more visitors of some ME at level ℓ *are* in T_m at level $\ell + 1$. By assumption that at most one process enters a leaf ME from any direction, $\ell > 0$.

Then for all $\ell' < \ell$, we have for all ME$'$ at level ℓ' in tree T_m that no two different visitors $p, q \in \{0, \ldots, S-1\}$ can be in T_m at level $\ell' + 1$. Then there at most two processors p, q, visitors of ME, concurrently accessing ME: one with $\beta = 0$, the other with $\beta = 1$. Now the original proof of mutual exclusion can be applied to show that at most one process can win [PF77]. This contradicts that both are in T_m at level ℓ. ■

Lemma 7 *If p is stuck in round r in tree T_m at level ℓ, then there is another process $q \neq p$ in tree T_m at the start of the r-th round of p.*

Proof: Let ME be the mutual exclusion at level ℓ in tree T_m played by p. Let us write R_p for the variable written by p in ME, and R'_p for the variable read by p and written by its opponent. Let β be the direction of p. As p is stuck in round r in tree T_m at level ℓ, its check of ME in round r returns *false*. At that time, then, $R'_p \neq nil$. Hence, there is a process q that writes R'_p while entering ME before p reads R'_p, and q does not release ME until after p reads R'_p. Such q must, using Lemma 6, have direction $1 - \beta$.

As p is stuck in round r, p checks ME in round $r-1$, which implies that during round r, p (and by Lemma 6 no other process either) does not write and thus change R_p, and that $R_p \neq nil$. Let r_p be the value of R_p during round r. Suppose q enters T_m after p starts round r. Then upon entering ME it reads $r_p \neq nil$ from R_p and thus sets R_q to $(1-\beta) \oplus r_p$. Then if p checks ME in round r, p reads $(1-\beta) \oplus r_p$ from R_q (which corresponds to R'_p) and evaluates $\beta \oplus (r_p \neq ((1-\beta) \oplus r_p))$ yielding *true* for arbitrary β and r_p, contrary to assumption that p is stuck in round r. Hence, q is in T_m before p starts round r. ■

The following proposition states that for different processes p and q, there are at most d trees that are tried by both of them. It is an easy consequence of the discussion in Section 4.1.

Proposition 8 *If $p \neq q$, then $\|N_p \cap N_q\| \le d$.*

Lemma 9 *For every process p, as long as p is not at level $\lceil \log S \rceil + 1$ in some tree T_m, in every round p advances to a higher level in at least $d(k-1)$ trees.*

Proof: Let r be an arbitrary round. By assumption at most $k-1$ processes other than p can be in any tree at the start of round r. Proposition 8 then implies that at the start of round r there is a process $q \neq p$ in at most $d(k-1)$ trees p is trying. By Lemma 7, p is stuck in round r in at most $d(k-1)$ trees it tries. As p plays $2d(k-1)$ trees per round, the lemma follows. ■

Theorem 10 *Protocol* FILTER *implements wait-free, long-lived renaming to $z2d(k-1)$ names in time $O(dk \log S)$ using $O(zdkS)$ variables.*

Proof: By Lemma 6, a name obtained by p is not obtained by any other processes for as long as p holds that name. Hence the protocol is a renaming protocol. As there are no restrictions on the execution of the protocol—except for the fact that at most k processes are contending for, or in fact have, a name—processes may repeatedly get and release a name. Hence the protocol is long-lived. There are D trees, each containing roughly $2S$ mutual exclusion blocks ME that each use 2 variables. By Section 4.1 we have $D = z2d(k-1)$, so FILTER uses $O(zdkS)$ variables. The maximum number of shared variable accesses performed by any process before it gets a name is computed as follows. Because a process p plays in at most $2d(k-1)$ trees, it can advance at most $2d(k-1)\lceil \log S \rceil$ times. Suppose a process p executes a *Check* $3d(k-1)$ times. If it loses $2d(k-1)$ or more of them, then there is a round in which p is stuck in at least $d(k-1)$ trees it is trying, contradicting Lemma 9. We conclude that for every $3d(k-1)$ *Check*'s performed, a process wins at least $d(k-1)$ of them. Then after at most $6d(k-1)\lceil \log S \rceil$ calls of *Check*, each taking 1 shared access, process p obtains a name. A process will *Enter* the $2d(k-1)\lceil \log S \rceil$ mutual-exclusions at most once, at the cost of 4 shared accesses. Clearly, releasing all played mutual exclusion blocks takes no more time than entering them. This shows that the protocol is wait-free, with time complexity $O(dk \log S)$. ■

4.4 Using FILTER

In this section, we discuss how the FILTER protocol can be used. First, the choice of d and z given k and S influences its time complexity, its space complexity, and the size of the destination name space. To demonstrate this, we consider various examples of k and S, and show how d and z can be chosen to satisfy the requirements outlined in Section 4.1. At the end of this section, we discuss how instances of filter (parameterized by d and z) can be combined with each other, and with other long-lived renaming protocols, to successively reduce the size of the destination name space.

In the following paragraphs, we consider various relationships between k and S. For each case, we give nearly optimal choices for d and z that satisfy the requirements set out in Section 4.1 and give the space and time complexity of the resulting protocol, along with the size of the destination name space.

$S \leq c^k$: Let $d = k$ and z prime such that $2k(k-1)+c \leq z \leq 4k(k-1)+2c$. This satisfies (1) and (2) and yields $D \leq [4k(k-1)+2c]2k(k-1) \leq 8k^4 + 4ck^2$. The time complexity of the resulting protocol is $O(k^3)$ and the space complexity is $O(k^4 c^k)$.

$S \leq 3^{k-1}$: If we set $d = (k-2)/2$ we get, by equation (2) in Section 4.1, $z \geq k^2 - 3k + 2$. As for any a there is a prime between a and $2a$ [Che52], we select $k^2 \leq z \leq 2k^2$. Then $z^{d+1} \geq (k^2)^{((k-2)/2)+1} = k^k$. As $k^k \geq 3^{k-1}$ if $k \geq 1$, this satisfies equation (1), and, by equation (3), $D \leq 2k^2(k-2)(k-1) \leq 2k^4$. The time complexity of the resulting protocol is $O(k^3)$ and the space complexity is $O(k^4 3^k)$.

$S \leq k^{\log^c k}$: Pick $d = \log^c k$ and z a prime such that $2k \log^c k \leq z \leq 4k \log^c k$. This again satisfies (1) and (2) and yields $D \leq 8k(k-1)\log^c k \log^c k$. The time complexity of the resulting protocol is $O(k \log^{2c+1} k)$ and the space complexity is $O(k^{2+\log^c k} \log^{2c} k)$.

$S \leq k^c$: Select $d = c$ and let z be a prime such that $2c(k-1) \leq z \leq 4c(k-1)$. This also satisfies (1) and (2) and yields $D \leq 4c(k-1)2c(k-1)$. The time complexity of the resulting protocol is $O(k \log k)$ and the space complexity is $O(k^{c+2})$.

$S \leq 2k^4$: Let us set $d = 3$. Then $z \geq 6(k-1)$ by equation (2), so let us pick $6k \leq z \leq 12k$ and z a prime. Then $z^{d+1} \geq (6k)^4 \geq 2k^4$ satisfying equation (1). For the size of the destination name-space we now find $D \leq 12k6(k-1) \leq 72k^2$. The time complexity of the resulting protocol is $O(k \log k)$ and the space complexity is $O(k^4)$.

This analysis shows that if the size of the source name space is polynomial in k, then FILTER renames to a name space of size $O(c^2k^2)$ in time $O(k \log k)$. Only if the size of the source name space becomes exponential in k will FILTER require $O(k^3)$ time. In all cases, the space complexity is never much bigger than S.

Renaming protocols can be nested in order to repeatedly reduce the size of the name space. To see this, observe that after acquiring a name from a particular long-lived renaming protocol, a process can then use the acquired name to access another long-lived renaming protocol. The latter protocol can have a source name space the same size as the destination name space of the former. In fact, several long-lived renaming protocols can be chained together in this fashion.

To see how this combining approach can be helpful, observe that for $S = k^{\log^c k}$ and $S = c^k$, applying FILTER twice yields $D \in O(k^2)$ with no asymptotic increase in the time or space complexity listed above. Unfortunately, Proposition 3.4 of Erdös *et al.* [EFF85] shows that multiple applications of protocol FILTER will not lead to a name space smaller than $k(k+1)/2$. To actually reach a name space of $k(k+1)/2$ one might apply protocol MA on the last name space obtained, at the expense of increasing the time complexity to $O(k^3)$.

For arbitrary values of S and k a destination name space of $k(k+1)/2$ can be achieved by combining SPLIT, the second and last instances of FILTER given above, and finally Moir and Anderson's protocol MA. This approach yields the following result.

Theorem 11 *Long-lived renaming from a source name space of size S to a destination name space of size $k(k+1)/2$ can be achieved with time complexity $O(k^3)$ and space complexity $O(k^4 \min(3^k, S))$.*

5 Concluding Remarks

As discussed previously, renaming to a smaller name space results in lower overhead. Thus, we are motivated to seek renaming protocols whose destination name spaces are as small as possible. Herlihy and Shavit [HS93] have shown that one-time renaming (and hence long-lived renaming) cannot be solved in a wait-free manner using atomic reads and writes unless $D \geq 2k-1$. For one-time renaming this bound is tight [BG93, MA94]; for long-lived renaming the best upper bound known is $D \in O(k^2)$, independent of whether the implementation is fast or not [MA94]. It would be interesting to close this gap, either by finding a long-lived renaming protocol with a smaller destination name space, or by showing that this is impossible. It would also be interesting to determine whether the fastness requirement has any influence on these bounds.

6 Acknowledgements

This work was partly carried out while J. Garay and M. Moir were visiting CWI. The authors thank the Center for its ambiance and hospitality.

References

[ABND+90] ATTIYA, H., BAR-NOY, A., DOLEV, D., PELEG, D., AND REISCHUK, R. Renaming in an asynchronous environment. *Journal of the ACM* **37**, 3 (1990), 524–548.

[AM94] ANDERSON, J. H., AND MOIR, M. Using k-exclusion to implement resilient, scalable shared objects. In *13th Ann. Symp. on Principles of Distributed Computing* (Los Angeles, CA, USA, 1994), pp. 141–150.

[BG93] BOROWSKY, E., AND GAFNI, E. Immediate atomic snapshots and fast renaming. In *12th Ann. Symp. on Principles of Distributed Computing* (Ithaca, N.Y., USA, 1993), pp. 41–51.

[BLS93] H. BUHRMAN, L. LONGPRÉ, AND E. SPAAN. Sparse reduces conjunctively to tally. In *8th Ann. Conf. Structure in Complexity Theory*, (San Diego, CA, USA, 1993), pp. 208–214. To appear in SIAM Journal on Computing 1995.

[BND89] BAR-NOY, A., AND DOLEV, D. Shared memory versus message-passing in an asynchronous distributed environment. In *8th Ann. Symp. on Principles of Distributed Computing* (Edmonton, Alberta, Canada, 1989), pp. 307–318.

[BNDKP91] BAR-NOY, A., DOLEV, D., KOLLER, D., AND PELEG, D. Fault-tolerant critical section management in asynchronous environments. *Information and Computation* **95**, 1 (1991), 1–20.

[Che52] CHEBYSHEV, L. Mémoire sur les nombres premiers. *Journal de Math.* **17** (1852), 366–390.

[Coh74] COHN, P. M. *Algebra Volume 1.* John Wiley & Sons, 1974.

[EFF85] ERDÖS, P., FRANKL, P., AND FÜREDI, Z. Families of finite sets in which no set is covered by the union of r others. *Israel Journal of Mathematics* **51**, 1–2 (1985), 79–89.

[HS93] HERLIHY, M., AND SHAVIT, N. The asynchronous computability theorem for t-resilient tasks. In *25th Ann. Symp. on Theory of Computing* (San Diego, CA, USA, 1993), pp. 111–120.

[MA94] MOIR, M., AND ANDERSON, J. H. Fast, long-lived renaming. In *8th Int. Workshop on Distributed Algorithms* (Terschelling, The Netherlands, 1994), pp. 141–155. To appear in Science of Computer Programming.

[PF77] PETERSON, G. L., AND FISCHER, M. J. Economical solutions for the critical section problem in a distributed system. In *9th Ann. Symp. on Theory of Computing* (Boulder, Colorado, USA, 1977).

[PPTV94] PANCONESI, A., PAPATRIANTAFILOU, M., TSIGAS, P., AND VITÁNYI, P. M. B. Randomized wait-free distributed naming. In *5th Int. Symp. on Algorithms and Computation* (Beijing, China, 1994), pp. 83–91.