Long-Lived Renaming Made Fast*

Harry Buhrman
Centrum voor Wiskunde en Informatica (CWI)
P.O. Box 94079, 1090 SB Amsterdam, The Netherlands
buhrman@cwi.nl

Juan A. Garay
IBM T. J. Watson Research Center
P.O. Box 704, Yorktown Heights, New York 10598, USA
garay@watson.ibm.com

Jaap-Henk Hoepman
CWI
P.O. Box 94079, 1090 SB Amsterdam, The Netherlands
jhh@cwi.nl

August 9, 1997

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* if (and only if) 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 $O(k \log k)$.

We also present an *adaptive* extension of this protocol for the case where the actual number of contending processors c is much smaller than the upper bound k on the contention. If S is polynomial in k, the time complexity of this protocol is $O(c \log k)$. Here, the cost is dominated by the actual contention.

Our results show, 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.

*Research partially supported by the Dutch foundation for scientific research (NWO) through NFI Proj. ALADDIN (contr. # NF 62-376). A preliminary version of this paper appeared in *Proc. 14th Ann. ACM Symp. on the Principles of Distributed Computing.*

1 Introduction

In the *one-time renaming problem* [BND89, ABND$^+$90, 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. To model contention, it is furthermore assumed that at most k out of n processes acquire or hold names concurrently. Then a renaming protocol is *fast* if 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.

A fast 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, Anderson and Moir [AM94] 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. Fast 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 usually much smaller. Suppose the time complexity of a task these processes are engaged in depends on the range of their identifiers. Then using a fast renaming protocol to reduce the size of that name space can dramatically improve performance.

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. The result is a long-lived renaming protocol with the same size of source name space as the first in the chain and the same size of destination name space as the last in the chain. The time and space complexity of the resulting protocol is the sum of the time and space complexities, respectively, of all the protocols in the chain. Once the size of the name space is made independent of the size of the original name space S, one can even employ non-fast protocols in the remainder of the chain. The resulting protocol will still be fast.

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]. Their 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 $O(kS)$.

The long-lived renaming problem is related to the *slotted l-exclusion* (or *l-assignment*) problem [BP89, ABND$^+$90, BNDKP91]. In the latter problem, l distinguishable copies of a shared resource must be shared among n processes, k of which may fail. Processes unable to obtain a resource must wait until a slot is freed. For this problem it is known that if $l < n$, then necessarily $l > 2k$ (and if $l \geq n$, processes trivially claim the resource with their own name). Complementing these lower bounds, if $l = 2k+1$, there exist protocols for both asynchronous, shared-memory, multiprocessing systems [BP89] and message passing systems [BNDKP91]. To be sure, these protocols are neither fast nor wait-free.

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 Moir and Garay [MG96] recently showed that, as a final back-end, the Burns and Peterson l-assignment protocol [BP89] can be used to solve fast and long-lived renaming to $2k-1$ names. They also present a fast long-lived renaming protocol that renames to $k(k+1)/2$ names in $O(k^3)$ time.

2 Summary of Results

In this paper, we present three long-lived renaming protocols: SPLIT, FILTER, and ADAPTIVE. SPLIT is a fast, $O(k)$ time, protocol that renames to $O(3^{k-1})$ names. FILTER is an ordinary long-lived renaming protocol that renames $O(3^{k-1})$ names to $O(k^2)$ names in $O(k^3)$ time. If the size S of the name space is polynomial in k, then protocol FILTER renames to $O(k^2)$ names in $O(k \log k)$ time. ADAPTIVE is a modified version of FILTER applicable when the actual number of contending processors c is much smaller than the upper bound k on the contention. If S is polynomial in k, the time complexity of ADAPTIVE is $O(c \log k)$.

The combination of SPLIT and FILTER yields a fast, long-lived renaming protocol to $O(k^2)$ names in $O(k^3)$ time. Usually the size S of the source name space is bounded from above by a function f of k. In practice we can assume $f \in O(3^k)$, so the first stage implemented by SPLIT is unnecessary. The size of the final name space can be reduced using other, non-fast, long-lived renaming protocols. We now present a brief description of the protocols.

The SPLIT protocol uses a collection of building blocks called *splitters*. Each splitter, if accessed by at most m processes concurrently, dynamically partitions these processes into three output sets, each of which contains at most $m-1$ 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 $O(k)$.

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 playing the associated mutual exclusion tournament. The winner of a tournament takes the associated name. This guarantees that no two processors ever hold the same name. Using (blocking) mutual exclusion in a wait-free protocol might seem counterintuitive. We present a modified version of Peterson and Fischer's mutual exclusion tournament trees [PF77]. This modification enables processes to compete for all names efficiently and "in parallel". Processors p are assigned sets N_p in such a way that "no set is covered by the union of $k-1$ others" [EFF85]. Hence, while process p is attempting to acquire a name, there is always a name $x \in N_p$ 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, blocking is avoided, and p can eventually acquire a name.

ADAPTIVE is a modified version of FILTER which adapts itself to the actual contention (the actual number of processes simultaneously trying to get a name) c instead of the upper bound on the contention k. The sets N_p used in protocol FILTER actually satisfy a stronger property: no subset of N_p of size $O(c)$ is covered by the union of $c-1$ other sets N_q. ADAPTIVE maintains an estimate $\hat{c}$ of the actual contention c. Initially $\hat{c}=1$. To get a name according to the ADAPTIVE protcol, processor p runs the FILTER protocol using only a subset of N_p of size $O(\hat{c})$. If p finds it does not make progress, it doubles its estimate of the contention and continues. The additional property of the sets N_p guarantees that as soon as the estimate of the contention $\hat{c}$ exceeds the true contention c, p will continue to make progres and will eventually get a name.

The remainder of the paper is organized as follows. Section 3 contains the definitions used in the paper. In Section 4, 5, and 6, we present the SPLIT, FILTER and ADAPTIVE protocols, respectively. Concluding remarks appear in Section 7.

3 Definitions

We consider an asynchronous, shared memory, multiprocessing environment in which n processes communicate through shared multi-writer 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 $S \geq n$.

A solution to the long-lived renaming problem consists of a wait-free implementation of operations $GetName_p$ — returning names in $\{0,\ldots,D-1\}$ — and $ReleaseName_p$ on a renaming object RN that is shared by n processes. For ease of discussion, define $name_p$ to equal:

? if p is busy executing $GetName_p$ or $ReleaseName_p$, or

$\perp$ if the last operation executed by p was $ReleaseName_p$, and p is not busy executing $GetName_p$ or $ReleaseName_p$, or

$x \in \{0,\ldots,D-1\}$ if the last operation executed by p was $GetName_p$ returning x, and p is not busy executing $GetName_p$ or $ReleaseName_p$.

Initially, $name_p = \perp$ for all p.

It is required that every process p executes the operations $GetName_p$ and $ReleaseName_p$ alternately. $GetName_p$ can only be called if $name_p = \perp$, and $ReleaseName_p$ can only be called if $name_p \in \{0,\ldots,D-1\}$. Given that at most k processes concurrently access RN, in other words

$$(\#p :: name_p \neq \perp) \leq k \ ,$$

any solution to the long-lived renaming problem must guarantee that the following assertion is an invariant:

$$p \neq q \text{ and } name_p \in \{0,\ldots,D-1\} \text{ implies } name_p \neq name_q \ .$$

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 := to denote assignment, *name* to denote local variables, NAME to denote shared variables, and **keyw** for keywords. Each labelled statement in our protocols 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 $l(s)$ 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 (where $s[1]$ 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.

4 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 *splitter* similar to the one employed by Moir and Anderson [MA94]. Processes accessing our splitter are dynamically partitioned into three output sets, denoted -1, 0 and 1. The splitter ensures that, if at any time at most m processes access a splitter concurrently, then each output set contains at most $m-1$ processes at any time.

The splitter is much like a railway junction that splits an incoming track in three disconnected outgoing tracks. Suppose that there are never more than m trains that have crossed the junction and have not returned yet. Then the workings of the junction guarantees that no more than $m-1$ trains are found on each of the three outgoing tracks -1, 0 and 1.

We start defining the splitter formally, and then show how it is used in the SPLIT protocol to implement long-lived renaming. Finally, in Sect. 4.1, we present an implementation of the splitter itself.

Splitter S consists of wait-free implementations of the two operations $\mathit{Enter}_p(\mathsf{S})$ — returning -1, 0, or 1 — and $\mathit{Release}_p(\mathsf{S})$. For ease of discussion, define $\mathit{out}_p(\mathsf{S})$ to equal :

? if p is busy executing $\mathit{Enter}_p(\mathsf{S})$ or $\mathit{Release}_p(\mathsf{S})$, or

$\perp$ if the last operation executed by p was $\mathit{Release}_p(\mathsf{S})$, and p is not busy executing $\mathit{Enter}_p(\mathsf{S})$ or $\mathit{Release}_p(\mathsf{S})$, or

$x \in \{-1, 0, 1\}$ if the last operation executed by p was $\mathit{Enter}_p(\mathsf{S})$ returning x, and p is not busy executing $\mathit{Enter}_p(\mathsf{S})$ or $\mathit{Release}_p(\mathsf{S})$.

Initially, $\mathit{out}_p(\mathsf{S}) = \perp$ for all p.

It is required that p executes $\mathit{Enter}_p(\mathsf{S})$ and $\mathit{Release}_p(\mathsf{S})$ on S alternately, starting with $\mathit{Enter}_p(\mathsf{S})$. Given that at most m processes concurrently access S, in other words

$$(\#p :: \mathit{out}_p(\mathsf{S}) \neq \perp) \leq m \ ,$$

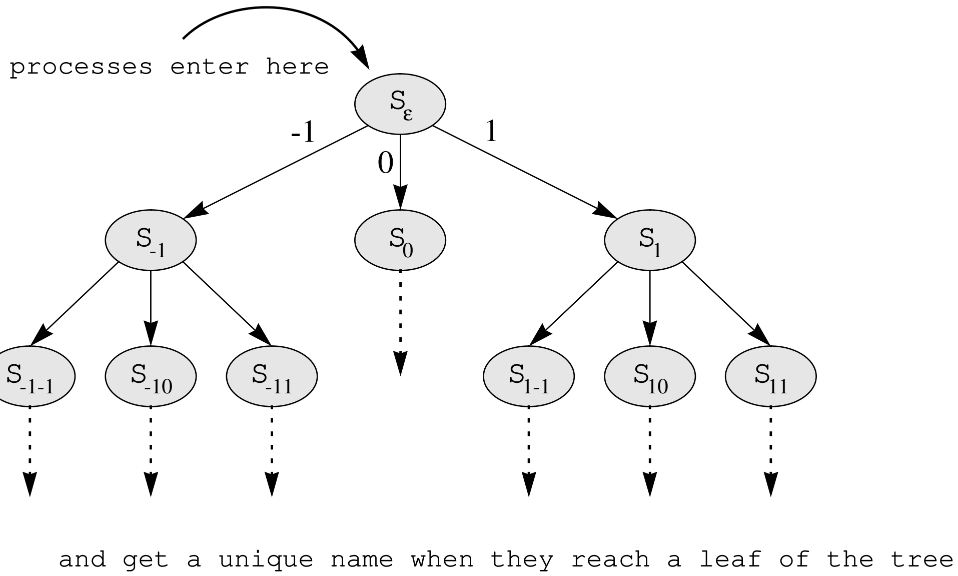

Figure 1: The tree of splitters used in protocol SPLIT.

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

operation *$GetName_p$*
 $s := \varepsilon$;
 for $\ell := k - 1$ **downto** 1
 do $e := Enter_p(\mathsf{S}_s)$;
  $s := (s\, e)$; (∗ Append e to s ∗)
 return $\overline{s}$; (∗ Compute name from path ∗)

operation *$ReleaseName_p$*
 for $\ell := 1$ **to** $k - 1$
 do $Release_p(\mathsf{S}_s)$;
  $s := s[1 : l(s) - 1]$; (∗ Discard last label in s ∗)

Protocol 4.1: Renaming protocol SPLIT for process p.

the implementation of splitter S must guarantee that the following assertion is an invariant:

$$(\forall d \in \{-1, 0, 1\} :: (\#p :: out_p(\mathsf{S}) = d) < m) \ .$$

To rename k processes, protocol SPLIT, shown in Prot. 4.1, conceptually uses a tree of depth $k - 1$ of splitters as shown in Figure 1. Each interior node has three children — connected to each of the three outputs -1, 0, and 1. Each splitter in the tree is uniquely labelled by a string corresponding to the sequence of outputs needed to reach it when starting at the root. The splitter at the root is labelled S_ε. The leafs of the tree are nodes connected to the outputs of the splitters at level $k-1$. They are labelled by strings of length $k-1$, corresponding to the sequence of outputs needed to leave the treat over that leaf.

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 node at level ℓ, p enters the splitter corresponding to that node. The output value returned by that splitter selects the next node at level $\ell+1$ to visit. The label s of the leaf reached is used to compute the name $\overline{s}$ 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 splitter guarantee that no other process is currently holding the same name. To release a name, a process simply releases all splitters it accessed in acquiring that name, in reverse order.

We now prove the correctness of protocol Split.

Lemma 4.1 *In protocol* Split*, the following assertion is an invariant for all* $s \in \{-1, 0, 1\}^{\leq k-2}$,

$$(\# p :: out_p(\mathsf{S}_s) \neq \perp) \leq k - l(s) \ .$$

Proof: By induction on $l(s)$. For $l(s) = 0$ (i.e., $s = \varepsilon$), 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.

$$(\# p :: out_p(\mathsf{S}_t) \neq \perp) \leq k - l(t) \ .$$

We assume that the splitter S_t is correct. Therefore, the correctness condition for the splitter implies that the following property is an invariant.

$$(\# p :: out_p(\mathsf{S}_t) = s[m+1]) \leq k - l(t) - 1 \ .$$

By the protocol, $out_p(\mathsf{S}_s) \neq \perp$ implies $out_p(\mathsf{S}_t) = s[m+1]$. Therefore

$$(\# p :: out_p(\mathsf{S}_s) \neq \perp) \leq k - l(s)$$

is an invariant. ◁

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

Theorem 4.2 *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 splitter S can be implemented in a wait-free manner, it is easy to see that SPLIT is wait-free. Moreover, the splitter implementation presented in Sect. 4.1 performs at most 9 shared variable accesses per operation. As each operation of SPLIT accesses $k-1$ splitters, the time complexity of SPLIT is $O(k)$.

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 $\overline{s} \neq \overline{t}$. Let p's name be $\overline{s}$, and let $s = (t\, d)$. Then p accessed splitter S_t at level $k-1$ returning d. By Lemma 4.1, $(\#p :: out_p(\mathsf{S}_t) \neq \perp) \leq 2$ is an invariant. Now the correctness condition for splitter S_t implies $(\#p :: out_p(\mathsf{S}_t) = d) \leq 1$. Hence, distinct processes do not concurrently hold the same name. $\lhd$

4.1 Implementing the splitter

In this section we present a wait-free implementation of the splitter S defined and used in protocol SPLIT in the previous section. It partitions processes into three sets, -1, 0, and 1. In order to join a set, process p calls $Enter_p(\mathsf{S})$ and joins the output set associated with the value returned. Later, p leaves that set by calling $Release_p(\mathsf{S})$[1]. As stated earlier, the implementation is required to ensure that, provided at most m processes concurrently access the splitter, no output set contains more than $m-1$ processes at any time.

Informally the idea is as follows. While accessing the splitter, 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 not all processes are 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 splitter 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 *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 $m-1$ processes at any time. Below, we describe the splitter implementation in more detail, before giving a correctness proof.

The splitter implementation appears in Prot. 4.2. LAST stores the identity of the last process to enter the splitter. Process p writes its identity to LAST upon entry, and reads it again at line 7 before deciding on a return value. If p reads $\mathsf{LAST} \neq p$ (i.e., p detects "interference" from another process), then p returns 0, otherwise p returns *dir* (which is easily shown to be non-zero). Observe that if m processes are inside the splitter, then the last process q to assign LAST reads $\mathsf{LAST} = q$, and therefore does not detect any interference. Thus, if m processes are inside the splitter, then at least one of the processes is not in output set 0.

[1] In the remainder of this section we omit the suffix (S), because we are interested in one splitter from now on.

Shared multi-writer variables:

```
    LAST ∈ {0, ..., S − 1} ;
    ADVICE[1] ∈ {⊥, −1, 1} ; initially 1
    ADVICE[2] ∈ {−1, 1} ; initially 1
```

Local variables (static):

```
    dir ∈ {−1, 1} ;
    adv2 ∈ {true, false} ;
```

```
operation Enter_p
1    LAST := p ;
2    dir := ADVICE[1] ;
     if dir = ⊥
3      then dir := ADVICE[2] ;
4    ADVICE[1] := −dir ;
5    adv2 := (LAST = p) ;
     if adv2
6      then ADVICE[2] := −dir ;
7    if LAST = p
       then return dir ;
       else return 0 ;
8      (* Working Section *)
```

```
operation Release_p
9    if LAST = p
10     then ADVICE[1] := dir ;
     if ¬adv2
11     then ADVICE[1] := ⊥ ;
12     (* Remainder Section *)
```

Protocol 4.2: Process p's code for splitter S.

To see that at most $m-1$ processes are in output set 1 at any time is more complicated (the case for set -1 is symmetric). First, note that if m processes are in output set 1, then each process p read $\mathsf{LAST} = p$ at line 7. This is only possible if m 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 entering, 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 m 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 $m-1$ processes already in an output set after q executes step 7, at most one process accesses the splitter concurrently before p executes 1. Because none of the remaining $m-1$ 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 splitter. We now present a formal correctness proof for the splitter.

4.2 Correctness of the splitter

In this section, we assume that $(\#p :: out_p \neq \bot) \leq m$ holds throughout a fixed, but arbitrary execution of the splitter implementation shown in Prot. 4.2. To show that the implementation is correct, we prove that, for each $d \in \{-1, 0, 1\}$, the following assertion is also an invariant.

$$(\#p :: out_p = d) < m \ .$$

In the following, we use p to denote both process p and an invocation of $Enter_p$ and the corresponding $Release_p$. In the rare instances where we consider different invocations by the same process, we distinguish the invocations (e.g., using p and p').

Recall that all accesses to shared variables are atomic. Each execution of the protocol therefore corresponds to a total order over all its atomic actions. As usual, we write $\Rightarrow$ for this total order. We use out_p to denote the value for *Enter* returned by invocation p; $p.1$ denotes the atomic action of executing line 1 by invocation p; the predicate $p@1$ means that the next line to be executed by process p is 1; $p_i@8\text{–}10$ means that the next line to be executed by process p lies in the range $8, \ldots, 10$. Finally, dir_p denotes the value of local variable dir held by processor p during this particular invocation.

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

Claim 4.3 *For an arbitrary invocation p, $out_p = 0$ iff there exists an invocation q such that $p.1 \Rightarrow q.1 \Rightarrow p.7$.*

Proof: If there exists a q such that $p.1 \Rightarrow q.1 \Rightarrow p.7$, then at line 7, p reads $\mathsf{LAST} \neq p$ and so $out_p = 0$, else p reads $\mathsf{LAST} = p$ and then $out_p \neq 0$. $\triangleleft$

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

Lemma 4.4 *Let $d \neq 0$. Suppose that at some point there are $m-1$ invocations $p_1, \ldots, p_{m-1}$ with $p_i@8\text{–}10$ and $out_{p_i} = d$, and there is an invocation q with $q@4$ and $p_i.7 \Rightarrow q.1$ for $1 \leq i \leq m-1$. Then $dir_q = -d$.*

Proof: Without loss of generality, assume $p_i.1 \Rightarrow p_{i+1}.1$ for $1 \leq i \leq m-2$. By Claim 4.3 and by assumption $out_{p_i} \neq 0$ for $1 \leq i \leq m-1$, we also have $p_i.7 \Rightarrow p_{i+1}.1$ for $1 \leq i \leq m-2$. In other words, $p_1, \ldots, p_{m-1}$ must have entered sequentially. Towards a contradiction, suppose that at time t for the first time the conditions of the lemma hold, but $dir_q = d$.

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_{m-1} did (case 1, 2, and 3 below), or not (case 4 below). If some invocation r writes $\mathsf{ADVICE}[1]$ after p_{m-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 r must finish before q starts, because $(\#p :: out_p \neq \bot) \leq m$. Hence if r executes line 10 or 11 then $r.10, r.11 \Rightarrow q.1$. Also in these three cases, as $out_{p_{m-1}} \neq 0$, by Claim 4.3 we have either $r.1 \Rightarrow p_{m-1}.1$ (case a) or $p_{m-1}.7 \Rightarrow r.1$ (case b).

1. For some r, $p_{m-1}.4 \Rightarrow r.10 \Rightarrow q.2$. Then q reads $\mathsf{ADVICE}[1] = dir_r = d$. If at line 9, $\mathsf{LAST} = r$ then also at line 7 and 5. Hence $out_r = d$. But then at a time $t' < t$ we have an earlier bad case contrary to assumption.

For if $p_{m-1}.7 \Rightarrow r.1$ (case b) then at time t' where $r@4$ we have $dir_r = d$ and still $p_i@8$–10 and $out_{p_i} = d$ for $i = 1, \ldots, m-1$.

And if $r.1 \Rightarrow p_{m-1}.1$ (case a, the only other possibility, see above) as $\mathsf{LAST} = r$ at line 9 we must have $r.9 \Rightarrow p_{m-1}.1$. But then at time t' where $p_{m-1}@4$ we have $dir_{p_{m-1}} = d$ by assumption that $out_{p_{m-1}} = d$ and still $p_i@8$–10 and $out_{p_i} = d$ for $i = 1, \ldots, m-2$ and (because $p_{m-1}.4 \Rightarrow r.10$ by assumption) also $r@8$–10 and $out_r = d$.

2. For some r, $p_{m-1}.4 \Rightarrow r.11 \Rightarrow q.2$. 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_{m-1}—because p_{m-1} sees $\mathsf{LAST} = p_{m-1}$ at 7 by assumption that $out_{p_{m-1}} \neq 0$, surely p_{m-1} sees $\mathsf{LAST} = p_{m-1}$ at 5 and hence executes $p_{m-1}.6$. But then $out_{p_{m-1}} = -d$, a contradiction.

   To see why p_{m-1} is the last to write $\mathsf{ADVICE}[2]$, let us suppose to the contrary that there is a r' with $p_{m-1}.6 \Rightarrow r'.6 \Rightarrow q.3$. 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 $(\#p :: out_p \neq \perp) \leq m$ then also $r'.6 \Rightarrow r.1$, and using $p_{m-1}.6 \Rightarrow r'.6$ we get $p_{m-1}.1 \Rightarrow r.1$. Again by $(\#p :: out_p \neq \perp) \leq m$ then r reads $\mathsf{LAST} = r$ at line 5 thus setting $adv2_r = true$, contrary to the assumption that r executes step 11.

3. For some $r \neq p_{m-1}$, $p_{m-1}.4 \Rightarrow r.4 \Rightarrow q.2$. As $(\#p :: out_p \neq \perp) \leq m$, if $p_{m-1}.7 \Rightarrow r.1$ (case b) then r reads $\mathsf{LAST} = r$ at line 9 and executes line 10. But then $r.10 \Rightarrow q.2$; this is case 1 above.

   Now consider $r.1 \Rightarrow p_{m-1}.1$ (case a, the only other possibility, see above). By $p_{m-1}.4 \Rightarrow r.4$ we get $r.1 \Rightarrow p_{m-1}.1 \Rightarrow r.5$, 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_{m-1} writes there (i.e., none of the above cases) at $p_{m-1}.4$ (because $p_{m-1}@8$–10, p_{m-1} did not execute line 10 yet). As $out_{p_{m-1}} = d$ this must be $-d$, a contradiction.

This completes the proof of Lemma 4.4. $\triangleleft$

Theorem 4.5 *For each $d \in \{-1, 0, 1\}$, the following assertion is an invariant.*

$$(\#p :: out_p = d) \leq m-1 \ .$$

Proof: Suppose for $d = 0$ the theorem does not hold. Then at some point $(\#p :: out_p = 0) = m$. 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 $(\#p :: out_p \neq \perp) \leq m$ there is no invocation r such that $q.1 \Rightarrow r.1 \Rightarrow q.7$. Hence by Claim 4.3, $out_q \neq 0$, a contradiction.

Towards a contradiction in the case where $d = -1$ or $d = 1$, suppose $(\#p :: out_p = d) = m$ at some point. Then similar to the proof of Lemma 4.4, by Claim 4.3 $p_1, \ldots, p_m$ must have entered sequentially, i.e., $p_i.7 \Rightarrow p_{i+1}.1$ for $1 \leq i \leq m-1$. Then at $p_m@4$ we have $p_i@8$–10 and $out_{p_i} = d$ for $1 \leq i \leq m-1$ and hence by Lemma 4.4 $dir_{p_m} = -d$. This contradicts $out_{p_m} = d$. $\triangleleft$

5 Reducing the Name Space

Protocol SPLIT on its own implements fast, long-lived renaming. However, the size of the resulting name space is 3^{k-1}, which is unacceptably high. We are therefore motivated to find long-lived renaming protocols that can reduce the size of the name space. Protocol FILTER, to be presented in this section, achieves exactly that.

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). To acquire a name m, a process competes in the mutual exclusion tree T_m associated with that name. This immediately guarantees that no two processes ever hold the same name simultaneously.

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 allowed for by using a modified version of Peterson and Fischer [PF77] mutual exclusion tournament 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], and in [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 5.1 below explains this hashing technique in detail. Then, in Sect. 5.2, we present the modified mutual exclusion tree and the FILTER protocol. Correctness of the protocol is proven in Sect. 5.3. Finally, in Sect. 5.4, we discuss the performance of FILTER under varying assumptions on the relationship between S and k.

5.1 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 must be assigned a polynomial $Q_p(x) = a_d x^d + \cdots a_1 x + a_0$ of degree d over field $\mathsf{GF}(z)$ (where $0 \le a_i < z$, and multiplication and addition are performed modulo z), such that the polynomials assigned to distinct processes differ in at least one coefficient. If

$$S \le z^{d+1} , \tag{1}$$

then we can assign to each process p the polynomial with for each i, $0 \le i \le d$, $a_i = (p \text{ div } z^i) \bmod z$. In other words, the i-th coefficient equals the i-th digit in the z-ary expansion of p's name.

Define $n_p(x) \triangleq zx + Q_p(x)$ (not doing arithmetic modulo z here) and $N_p \triangleq \{n_p(0), \ldots, n_p(2d(k-1)-1)\}$. Note that for arbitrary $p \in \{0, \ldots, S-1\}$ and $x \in \mathsf{GF}(z)$, $Q_p(x) < z$. It follows

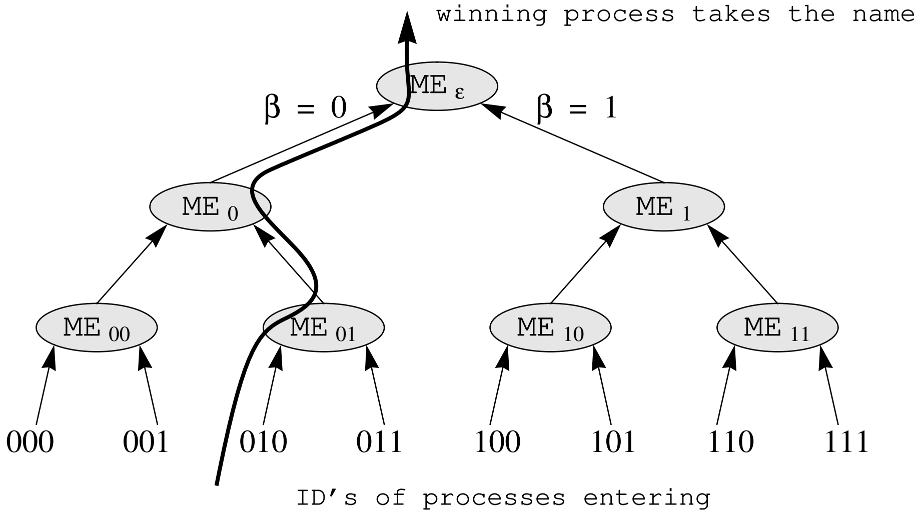

Figure 2: A tournament tree for 8 processes.

that $n_p(x) = n_q(y)$ if and only if $x = y$ 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 $x < z$ such that $Q_p(x) = Q_q(x)$ [Coh74]. Recall that for $p \neq q$, Q_p and Q_q are distinct. Then

$$z \geq 2d(k-1) \tag{2}$$

implies $|N_p \cap N_q| \leq d$. This guarantees that there are at most d names competed for by both of any pair of two processes.

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. Therefore, the largest name competed for by any p is the maximum over all p and all x ranging over $0 \leq x \leq 2d(k-1)-1$ of $n_p(x) = zx + Q_p(x)$. Recall that $Q_p(x)$ is bounded by z. Hence no process competes for a name larger than $z2d(k-1)$, which happens to be less than or equal z^2 by (2). 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 Sect. 5.4, 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.

Shared multi-writer variables:
$\mathsf{ME.R}[0], \mathsf{ME.R}[1] \in \{\perp, 0, 1\}$;

operation *Enter*(ME, β)
if $\mathsf{ME.R}[1-\beta] = \perp$ **then** $\mathsf{ME.R}[\beta] := 1$
else $\mathsf{ME.R}[\beta] := \beta \oplus \mathsf{ME.R}[1-\beta]$;
if $\mathsf{ME.R}[1-\beta] \neq \perp$ **then** $\mathsf{ME.R}[\beta] := \beta \oplus \mathsf{ME.R}[1-\beta]$;

operation *Check*(ME, β)
return $(\mathsf{ME.R}[1-\beta] = \perp) \vee$
$\beta \oplus (\mathsf{ME.R}[\beta] \neq \mathsf{ME.R}[1-\beta]))$;

operation *Release*(ME, β)
$\mathsf{ME.R}[\beta] := \perp$;

Protocol 5.1: Implementation of the 2-process mutual exclusion block ME.

5.2 Protocol Filter in detail

For each name $m \in \{0, \ldots, D-1\}$, FILTER uses a binary mutual exclusion tournament tree T_m to guarantee exclusive access name m for the process that wins T_m. Each tournament tree has $\lceil \log S \rceil$ levels. The leaves are at level 1 and the root is at level $\lceil \log S \rceil$. An example for $S = 8$ appears in Figure 2. Each node in tree T_m is a distinct two-process mutual exclusion block (ME) with two "input edges" labelled 0 for left and 1 for right. A process starts playing in a tournament tree by entering a mutual exclusion at level 1 over one of its two inputs. Each process enters a tree over a different input. Only if p reaches the critical section of a mutual exclusion at level ℓ, it advances to the parent at level $\ell + 1$. We write $\mathsf{ME}^m_{p,\ell}$ for the ME block in tree T_m played by process p at level ℓ.

As in protocol SPLIT, each mutual exclusion block in a tournament tree is uniquely labelled by a string corresponding to the sequence of input edges traversed to reach it when starting at the root (i.e., , when traversing the tournament tree in the reverse direction). The mutual exclusion block at the root is labelled ε. Note that there are $2^{\lceil \log S \rceil} \geq S$ "inputs" to the leaves of the tree. Consider p's name to be a binary string of length $\lceil \log S \rceil$ (padded in front by 0's). In FILTER, p enters tree T_m at the ME block at level 1 with label $p[2 : \lceil \log S \rceil]$ over input $p[1]$. Then no other process will enter this ME block over the same input, as required. And in general, given the above labelling of the ME blocks, it follows that $\mathsf{ME}^m_{p,\ell}$ has label $p[\ell + 1 : \lceil \log S \rceil]$. Process p enters $\mathsf{ME}^m_{p,\ell}$ over input $p[\ell]$.

To allow processes to compete in several tournament trees "in parallel", the two-process ME block used in these trees is, in fact, 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 implementation of the two-process ME block is presented in Prot. 5.1. In this protocol, we use $\oplus$ to denote *exclusive or*, and $\vee$ to denote disjunction. We set *true* $= 1$ and *false* $= 0$.

repeat (∗ this starts a new round ∗)
  forall $m \in N_p$ **do**
    if p did not enter tree m yet
      then Enter tree T_m at the appropriate leaf.
    Let ℓ be the last entered level of tree T_m.
    while $\ell \neq \lceil \log S \rceil$ and checking ME at level ℓ in tree T_m returns true
    do Let $\ell := \ell + 1$ and enter ME at level ℓ.
    if $\ell = \lceil \log S \rceil$ and checking ME at level ℓ in tree T_m returns true
      then return name m
until a name was found

Protocol 5.2: How p gets a name in the FILTER protocol.

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 ME block 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. Each mutual exclusion block ME uses a separate multi-writer register ME.R, for which we write R if no confusion can arise.

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 $\beta = 1$.

Having described the tournament trees and the mutual exclusion blocks they use, we can now explain protocol FILTER. An informal description how process p obtains a name appears in Prot. 5.2. The details appear in Prot. 5.3. Process p competes "in parallel" in all the tournament trees associated with names in the set N_p (as defined in Sect. 5.1 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, continuing at the last reached level in that tree. This is repeated until a name has been acquired (by reaching the critical section of the root of the tree for that name), or all names have been tried. In the latter case, process p returns to the tree for the first name in N_p. This is repeated until p acquires a name.

To release a name (only shown in Prot. 5.3), 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 these trees 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 block ensures that p is able to proceed at least one level

```
operation GetName_p
    for m ∈ N_p do level_p[m] := 0 ;                         (* initially, no tree is entered yet *)
    repeat                                                              (* this starts a new round *)
        for m ∈ N_p do                                              (* try all names in any order *)
        if TryTree_p(m) then return m;
    until false ;

operation TryTree_p(m)
    if level_p[m] = 0                                                    (* if p hasn't entered T_m yet *)
      then Enter_p(ME^m_{p,1}, p[1]) ; ℓ := 1                                  (* enter T_m at level 1 *)
      else ℓ := level_p[m] ;                       (* resume where you left in previous round *)
    while (ℓ ≠ ⌈log S⌉) ∧ Check_p(ME^m_{p,ℓ}, p[ℓ])                       (* increase level until *)
      do ℓ := ℓ + 1 ; Enter_p(ME^m_{p,ℓ}, p[ℓ]) ;                                  (* ME is blocked *)
    level_p[m] := ℓ ;                                                        (* save reached level *)
    return (ℓ = ⌈log S⌉ ∧ Check_p(ME^m_{p,ℓ}, p[ℓ]) ;

operation ReleaseName_p
    for m ∈ N_p
    do for ℓ = level_p[m] to 1                                   (* release all claimed ME blocks *)
        do Release_p(ME^m_{p,ℓ}, p[ℓ]) ;                                      (* (order is relevant) *)
```

Protocol 5.3: Precise implementation of protocol FILTER.

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.

5.3 Correctness proof for Filter

We say that a process p is *in tree T_m at level ℓ* if it has started entering $\mathsf{ME}^m_{p,\ell}$ and has not finished releasing $\mathsf{ME}^m_{p,\ell}$. 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 (i.e., if $name_p = m$). We say that a process p is *in tree T_m* if it is in T_m at some level. Recall the assumption 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 the first name in N_p. We say a process p is *stuck at round r in tree T_m at level ℓ* if in two successive rounds $r-1$ and r, checking $\mathsf{ME}^m_{p,\ell}$ at level ℓ in tree T_m returns *false*.

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

Lemma 5.1 *For all ℓ with $1 \le \ell \le \lceil \log S \rceil$ and all $p, q \in \{0, \dots, S-1\}$, if $\mathsf{ME}^m_{p,\ell} = \mathsf{ME}^m_{q,\ell}$, then p and q cannot be in T_m at level $\ell + 1$ simultaneously.*

Proof: Suppose the lemma does not hold. Towards a contradiction pick the minimal ℓ for which there are p, q with $\mathsf{ME}^m_{p,\ell} = \mathsf{ME}^m_{q,\ell}$ that *are* in T_m at level $\ell + 1$ simultaneously.

Then for all ℓ', such that $1 \le \ell' < \ell$, no $p, q \in \{0, \dots, S-1\}$ with $\mathsf{ME}^m_{p,\ell'} = \mathsf{ME}^m_{q,\ell'}$ are in T_m at level $\ell' + 1$ simultaneously. Also, by assumption, at most one process enters a ME at level 1 from any direction. Hence at most two processors p, q, with $\mathsf{ME}^m_{p,\ell} = \mathsf{ME}^m_{q,\ell} = \mathsf{ME}$ concurrently access 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 the assumption that both are in T_m at level $\ell + 1$. ◁

Lemma 5.2 *If p is stuck at 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 $\mathsf{ME} = \mathsf{ME}^m_{p,\ell}$, 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 $\beta = p[\ell]$, 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, $\mathsf{R}'_p \neq \perp$. 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 5.1, 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 5.1 no other process either) does not write and thus change R_p, and that $\mathsf{R}_p \neq \perp$. 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 $r_p \neq \perp$ 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 Sect. 5.1.

Proposition 5.3 *If $p \neq q$, then $|N_p \cap N_q| \leq d$.*

Lemma 5.4 *In each round, every process p can advance to a higher level in at least $d(k-1)$ trees it is trying.*

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 5.3 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 5.2, 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 5.5 *Protocol* FILTER *implements wait-free and long-lived renaming to $z2d(k-1)$ names in time $O(dk \log S)$ using $O(zdkS)$ variables.*

Proof: By Lemma 5.1, 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, containing roughly $2S$ mutual exclusion blocks ME that each use 2 variables. By Sect. 5.1, Eq. (3) we have $D = z2d(k-1)$, so FILTER uses $O(zdkS)$ variables[2].

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 (or else it wins some tree). Suppose a process p executes a *Check* $4d(k-1)$ times. If it loses $3d(k-1)$ 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 5.4. (If it loses $3d(k-1)$ out of $4d(k-1)$, then $2d(k-1)$ of those were lost immediately, and $d(k-1)$ of those must belong to a single round. Because they were lost immediately, p is stuck at those $d(k-1)$ ME blocks.) We conclude that for every $4d(k-1)$ *Check*'s performed, a process wins at least $d(k-1)$ of them. Then after at most $8d(k-1)\lceil \log S \rceil$ 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 $O(dk \log S)$. $\triangleleft$

5.4 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 Sect. 5.1. 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 Sect. 5.1 and give the space and time complexity of the resulting protocol, along with the size of the destination name space.

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

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

$S \le k^{\log^c k}$: Pick $d = \log^c k$ and z a prime such that $2k \log^c k \le z \le 4k \log^c k$. This again satisfies Eq. (1) and Eq. (2). Substituting d and z in Eq. (3) yields $D \le 8k(k-$

[2] Actually, because processes contend only for a fraction of the names, certain mutual exclusion blocks may never be accessed. These blocks may safely be removed from the trees, but this will not significantly decrease the space complexity.

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 Eq. (1) and Eq. (2). Substituting d and z in Eq. (3) 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 Eq. (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 Eq. (1). In this case, for the size of the destination name-space we 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.

In the introduction, we stated that renaming protocols can be nested in order to repeatedly reduce the size of the name space. 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, Prop. 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 reach a name space of $2k-1$, one might apply Burns and Peterson l-assignment protocol [BP89] as in [MG96], at the expense of increasing the time complexity to $O(k^4)$.

For arbitrary values of S and k a destination name space of $2k-1$ can be achieved by combining SPLIT, the second and last instances of FILTER given above, and finally the Burns and Peterson l-assignment protocol [BP89].

6 Adapting to the Actual Contention

So far, our algorithms assume the existence of an upper bound on the contention, denoted by k. Their time complexity was expressed using this constant. The question is whether one can improve the time complexity of our algorithms for the case where the actual contention c is much smaller than k. In other words, whether one can derive long-lived renaming protocols that adapt to the actual contention. Ideally, one would like the time complexity to be independent of k altogether. In this section we develop protocol ADAPTIVE, which reduces the dependency on k to a single $\log k$ factor.

ADAPTIVE is a modified version of FILTER which adapts itself to the actual contention c instead of the upper bound on the contention k. If the actual contention is c, no more than $c-1$ other processes are busy acquiring, holding or releasing a name. The main idea is based on the observation that the sets N_p used in protocol FILTER actually satisfy a stronger property: no subset of N_p of size $O(c)$ is covered by the union of $c-1$ other sets N_q. In fact, if we return briefly to Sect. 5.1, we see that by $|N_p \cap N_q| \leq d$ at most $d(c-1)$ members of N_p are covered by the union of $c-1$ other sets N_q.

```
ĉ_p := 1                                   (* initialise estimate of contention to 1 *)
repeat (* this starts a new round *)
    i := 0 ; won_p := 0 ; lost_p := 0 ;
                        (* play the first ĉ_p trees until you've won more than you've lost *)
    while won_p ≤ lost_p and i < 2dĉ_p
    do m = n_p(i)                                        (* m is the i-th name in N_p *)
       if p did not enter tree m yet
          then Enter tree T_m at the appropriate leaf.
       Let ℓ be the last entered level of tree T_m.
       while ℓ ≠ ⌈log S⌉ and checking ME at level ℓ in tree T_m 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 T_m returns true
          then return name m                            (* p has won tournament T_m *)
       if p advanced in tree T_m at least once               (* update won_p and lost_p *)
          then won_p := won_p + 1
          else lost_p := lost_p + 1
       i := i + 1                                                (* try the next tree *)
    od
    if won_p ≤ lost_p                              (* estimate of contention was too low *)
      then ĉ_p = min(2 * ĉ_p, k − 1)                (* double the estimate of contention *)
until a name was found
```

Protocol 6.1: How p gets a name in the Adaptive protocol.

Each process in protocol Adaptive maintains an estimate $\hat{c}_p$ of the actual contention c. Initially $\hat{c}_p = 1$. To get a name according to the Adaptive protcol, processor p runs the Filter protocol using only the first $2d\hat{c}_p$ names of N_p. In each such round, p counts the number of trees in which it advanced at least once (in the variable won_p), and also counts the number of tree in which it lost immediately (in the variable $lost_p$). If p finds it does not make progress, i.e., if the number of trees advanced in at least once is less than or equal to the number of trees lost immediately ($won_p \leq lost_p$), it doubles its estimate of the contention and continues. The aforementioned property of the sets N_p guarantees that as soon as the estimate of the contention $\hat{c}_p$ exceeds the true contention c, p will continue to make progres and will eventually get a name. The details appear in Prot. 6.1.

6.1 Correctness proof for Filter

Let $c \leq k$ be an upperbound on the actual contention during the time that process p tries to acquire a name through the Adaptive protocol.

Lemma 6.1 *In each round, process p cannot advance to a higher level in at most $d(c-1)$ trees it is trying.*

Proof: Let r be an arbitrary round. By assumption at most $c-1$ processes other than p can be in any tree at the start of round r. Proposition 5.3 then implies that at the start of round r there is a process $q \neq p$ in at most $d(c-1)$ trees p is trying. By Lemma 5.2, p is stuck in round r in at most $d(c-1)$ trees it tries. $\triangleleft$

Theorem 6.2 *Protocol* ADAPTIVE *implements wait-free and long-lived renaming to $z2d(k-1)$ names in time $O(dc\log S)$, if the contention while acquiring a name is at most c.*

Proof: By Lemma 5.1, 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 $c-1$ processes are contending for, or in fact have, a name—processes may repeatedly get and release a name. Hence the protocol is long-lived.

The maximum number of shared variable accesses performed by any process before it gets a name is computed as follows.

By Lemma 6.1 and the protocol, process p will only play the first $2d(c-1)$ trees of N_p in each round. To get a name, p therefore needs to advance at most $2d(c-1)\lceil\log S\rceil$ times. After this many wins, p is guaranteed to have reached and won the root of a tree, and thus has acquired the associated name.

After each such round, either $won_p > lost_p$, or $\hat{c}_p$ is multiplied by two. To bound the number of times a tree is lost, consider two types of rounds: one after which $won_p > lost_p$, and the other after which $\hat{c}_p$ is multiplied by two. Note that once $\hat{c}_p \geq c$, $won_p > lost_p$ after each round.

The number of trees tried in rounds of the second type form a geometric series $1, 2, 4, \ldots 2d2^x$ with $\hat{c}_p = 2^x > c > 2^{x-1}$. Therefore, the number of times a tree was lost in the second group cannot exceed $8dc$.

Because p needs to advance at most $2d(c-1)\lceil\log S\rceil$ times before acquiring a name, the number of times a tree was lost in a round of the first type cannot exceed $2d(c-1)\lceil\log S\rceil$.

We conclude that after $4d(c-1)\lceil\log S\rceil + 8dc$ times trying a tree, p must have acquired a name. $\triangleleft$

7 Concluding Remarks

Fast and long-lived renaming, even to $2k-1$ names, is possible. If an upper bound on the size of the name space is known, the time complexity may be reduced considerably. For instance, if S is polynomial in k, one can implement long-lived renaming to $O(k^2)$ names in $O(k\log k)$ time. The technique used to achieve this, namely playing mutual exclusion tournament trees in parallel while using hashing techniques to avoid blocking, may be of independent interest.

So far we did not succeed in finding a protocol that is truly adaptive to the actual contention c. Instead, we present a protocol that, if S is polynomial in k, implements long-lived renaming to $O(k^2)$ names in $O(c\log k)$ time. The major challenge that remains is to remove this logaritmic dependence on k.

The SPLIT and FILTER algorithms have been implemented and tested by Sandra Moral Boandas [Boa97].

We acknowledge the help of Mark Moir in the earlier stages of this work.

References

[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), ACM Press, pp. 141–150.

[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.

[BND89] Bar-Noy, A., and Dolev, D. Shared-memory vs. message-passing in an asynchronous distributed environment. In *8th Ann. Symp. on Principles of Distributed Computing* (Edmonton, Alb., Canada, 1989), ACM Press, 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.

[Boa97] Moral, S. B. Reducción del espacio de nombres de procesos en el marco de los algoritmos distribuidos. Master's thesis, Universitat Politecnica de Catalunya, 1997.

[BG93] Borowsky, E., and Gafni, E. Immediate atomic snapshots and fast renaming. In *12th Ann. Symp. on Principles of Distributed Computing* (Ithaca, NY, USA, 1993), ACM Press, pp. 41–51.

[BLS93] Buhrman, H., Longpré, L., and Spaan, E. SPARSE reduces conjunctively to TALLY. In *Proc. Structure in Complexity Theory 8th annual conference* (San Diego, CA, USA, 1993), IEEE Computer Society Press, pp. 208–214.

[BP89] Burns, J. E., and Peterson, G. L. The ambiguity of choosing. In *8th Ann. Symp. on Principles of Distributed Computing* (Edmonton, Alb., Canada, 1989), ACM Press, pp. 145–157.

[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. P., and Shavit, N. The asynchronous computability theorem for t-resilient tasks. In *25th Ann. Symp. on Theory of Computing* (San Diego, CA, USA, 1993), ACM Press, 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), G. Tel and P. M. B. Vitányi (Eds.), Lect. Not. Comp. Sci. 857, Springer-Verlag, pp. 141–155. To appear in Science of Computer Programming.

[MG96] Moir, M., and Garay, J. A. Fast, long-lived renaming improved and simplified. In *10th International Workshop on Distributed Algorithms (WDAG '96)* (Bologna, Italy, 1996), Lect. Not. Comp. Sci. 1151, Springer-Verlag, pp. 287-303.

[PPTV94] Panconesi, A., Papatriantafilou, M., Tsigas, P., and Vitányi, P. M. B. Randomized wait-free naming. In *5th Int. Symp. on Algorithms and Computation* (Beijing, P. R. China, 1994), D.-Z. Du and X.-S. Zhang (Eds.), Lect. Not. Comp. Sci. 834, pp. 83–91.

[PF77] Peterson, G. L., and Fischer, M. J. Economical solutions for the critical section problem in a distributed system. In *9th Ann. Symp. on the Theory of Computing* (Boulder, CO, USA, 1977), ACM Press, pp. 91–97.