Polynomial and Adaptive Long-lived $(2k-1)$-Renaming*
(Extended Abstract)

Hagit Attiya and Arie Fouren

Department of Computer Science, The Technion, Haifa 32000, Israel

Abstract. In the long-lived M-renaming problem, processes repeatedly obtain and release new names taken from a domain of size M. This paper presents the first polynomial algorithm for long-lived $(2k-1)$-renaming. The algorithm is adaptive as its step complexity is $O(k^4)$; here k is the point contention—the maximal number of simultaneously active processes in some point of the execution. Polynomial step complexity is achieved by having processes help each other to obtain new names, while adaptiveness is achieved by a novel application of *sieves*.

1 Introduction

Distributed coordination algorithms are designed to accommodate a large number of processes, each with a distinct identifier. Often, only a few processes simultaneously participate in the coordination algorithm [19]. In this case, it is worthwhile to *rename* the participating processes [6, 21]: Before starting the coordination algorithm, a process uses getName to obtain a unique *new name*—a positive integer in the range $\{1, \ldots, M\}$; the process then performs the coordination algorithm, using the new name instead of its identifier; when the coordination algorithm completes, releaseName allows the name to be re-used later.

A renaming algorithm guarantees *adaptive name space* if M is a function of k, the maximal number of processes *simultaneously* obtaining new names; k is called the *point contention*. Obviously, M should be as small as possible, preferably *linear* in k; it is known that $M \geq 2k-1$ [15, 18]. For the renaming stage to be useful it must also have *adaptive step complexity*: The number of steps a process takes in order to obtain a new name (or to release it) is a function of k. Under this definition, getName is delayed only when many processes participate simultaneously. (Precise definitions appear in Section 2.)

This paper presents an adaptive algorithm for long-lived renaming, using read and write operations. In our algorithm, a process obtains a name in the range $\{1, \ldots, 2k-1\}$ with $O(k^4)$ steps. Thus, the algorithm's step complexity is a function of the maximal number of processes simultaneously active at some

* Supported by the fund for the promotion of sponsored research at the Technion. Due to space limitations, many details are omitted from this extended abstract; a full version of the paper is available through www.cs.technion.ac.il/~hagit/pubs.html.

point during the getName operation. This is the first long-lived renaming algorithm with optimal name space whose step complexity is polynomial in the point contention. All previous long-lived renaming algorithms providing linear name space have exponential step complexity [16].

The algorithm uses a building block called *sieve* [9]; sieves are employed in all known algorithms which adapt to point contention using read and write operations [1, 2, 4, 5]. A process tries to win in a sequence of sieves, one after the other, until successful in some sieve; when successful, the process suggests to reserve this sieve for some (possibly other) process. The sieve is reserved for one of the processes suggested for it. A process fails in a sieve only if some other process is inside this sieve; this is used to show that a process accesses sieve s only if $O(s)$ processes simultaneously participate.

Attiya et al. [6] introduce the *one-shot* renaming problem and present a $(2k-1)$-renaming algorithm for the message passing model; Bar-Noy and Dolev [10] translate this algorithm to the shared-memory model. The complexity of these algorithms is exponential [16]. Gafni [17] describes a one-shot $(2k-1)$-renaming algorithm with $O(n^3)$ step complexity. Borowsky and Gafni [12] present a one-shot $(2k-1)$-renaming algorithm with $O(N^2 n)$ step complexity.

Several adaptive renaming algorithms were suggested recently. Attiya and Fouren [7] present a $(6k-1)$-renaming algorithm with $O(k \log k)$ step complexity. Afek and Merritt [3] extend this algorithm to a $(2k-1)$-renaming algorithm with $O(k^2)$ step complexity. The $(2k-1)$-renaming algorithms of Gafni [17] and Borowsky and Gafni [12] can be made adaptive using an adaptive collect operation [8]. These algorithms are *one-shot* and adapt to the *total contention*: Their step complexity depends on the total number of operations performed so far, and does not decrease when the number of participating processes drops.

Burns and Peterson [15] present a long-lived $(2k-1)$-renaming algorithm whose step complexity is exponential [16].

Anderson and Moir [21] considered a system where many processes (N) may participate in an algorithm but in reality no more than $n \ll N$ processes are active. This paper and subsequent work [14, 20, 22] presented long-lived renaming algorithms where n is known in advance, culminating in a long-lived $(2k-1)$-renaming algorithm [20]. This algorithm employs Burns and Peterson's algorithm [15] and thus its step complexity is exponential in n.

Long-lived renaming algorithms that adapt to point contention were presented recently [1, 2, 9]. Some of these algorithms have $O(k^2 \log k)$ step complexity but they yield a name space of size $O(k^2)$; others provide linear name space (either $2k-1$ or $6k-1$ names), but their step complexity is exponential.

2 Preliminaries

In the *long-lived M-renaming* problem, processes $p_1, \ldots, p_n$ repeatedly acquire and release distinct names in the range $\{1, \ldots, M\}$. A solution supplies two procedures: getName returning a *new name*, and releaseName; p_i alternates between invoking $\mathsf{getName}_i$ and $\mathsf{releaseName}_i$, starting with $\mathsf{getName}_i$.

Consider α, an execution of a long-lived renaming algorithm; let α' be a finite prefix of α. Process p_i is *active* at the end of α', if α' includes an invocation of $\mathsf{getName}_i$ without a return from the matching $\mathsf{releaseName}_i$. A long-lived renaming algorithm should guarantee *uniqueness* of new names: Active processes hold distinct names at the end of α'.

The *point contention* (abbreviated *contention* below) at the end of α', denoted $\mathrm{PntCont}(\alpha')$, is the number of active processes at the end of α'. Consider a finite interval β of α; we can write $\alpha = \alpha_1\beta\alpha_2$. The contention during β, denoted $\mathrm{PntCont}(\beta)$, is the maximum contention in prefixes $\alpha_1\beta'$ of $\alpha_1\beta$. If $\mathrm{PntCont}(\beta) = k$, then k processes are simultaneously active at some point during β.

A renaming algorithm has an *adaptive name space* if there is a function M, such that the name obtained in an interval of $\mathsf{getName}$, β, is in the range $\{1, \ldots, \mathrm{M}(\mathrm{PntCont}(\beta))\}$.

A renaming algorithm has *adaptive step complexity* if there is a bounded function S, such that the number of steps performed by p_i in any interval of $\mathsf{getName}_i$, β, and in the matching $\mathsf{releaseName}_i$ is at most $\mathrm{S}(\mathrm{PntCont}(\beta))$. The contention during an interval is clearly bounded by n. Therefore, $\mathsf{getName}_i$ and $\mathsf{releaseName}_i$ terminate within a bounded number of steps of p_i, regardless of the behavior of other processes; hence, the algorithm is *wait-free*.

3 The Basic Sieve

This section describes the basic sieve [9], re-organized so it can be extended (in Section 4.2 below) to support reservations.

A sieve allows processes to obtain a view of the processes accessing the sieve concurrently, or no view (an empty view). There is a unique non-empty set of *candidates* which are seen by all processes getting a view. The sieve guarantees agreement on the information announced by candidates; this synchronizes the processes accessing the sieve and allows to exchange information in an adaptive manner.

A sieve has an infinite number of copies; at each point in the execution, processes are only "inside" a single copy of the sieve. The number of the current copy is monotonically incremented by one.

The sieve supports the following operations:

$\mathsf{read}(s.count)$: get the number of the current copy of sieve s.
$\mathsf{openFor}(s, c)$: returns **all**, if all operations can enter copy (s, c), and $\emptyset$ if no process can enter copy (s, c).
$\mathsf{enter}(s, c, info)$: enter copy (s, c), announcing *info*, returns the set of candidates, together with the information announced by them.
$\mathsf{exit}(s, c)$: leave the sieve and activate the next copy, $(s, c + 1)$, if possible.

To access sieve s, a process first reads the number of the current copy from $s.count$. The process enters the sieve using enter only if $\mathsf{openFor}$ returns **all**, and leaves the sieve using exit. A process accesses sieve s in the following order:

```
1. c = read(s.count)                              // find the current copy
2. if (openFor(s, c) == all) then                 // copy (s, c) is open
3.    enter(s, c, info)                           // enter (s, c) and announce info
...
4.    exit(s, c)                                  //leave the sieve
```

3.1 Implementation of the Basic Sieve

Sieves are implemented using ideas from the Borowsky-Gafni simulation [11, 13], modified to guarantee adaptive step complexity.

A sieve has an infinite number of copies. At each point of the execution, candidates are inside a single copy; these processes access the sieve simultaneously.

A process tries to get inside a sieve by checking if it is among the first processes to access the current copy of this sieve. It succeeds only if this copy is free (no other process is already inside it), and no candidate is inside the previous copy. If a process does not get inside the sieve, then some concurrent process is already inside the sieve. In this manner, the sieve "catches" at least one process: one of the processes which access the current copy enters the sieve. This property makes the sieve a useful tool in adapting to contention.

For each sieve there is an integer variable, *count*, indicating the current copy of the sieve; it is initially 1. There is an infinite number of copies for each sieve, numbered $1, 2, \ldots$. Each copy has the following data structures.

- An array $R[1, \ldots, N]$ of views; all views are initially empty. Entry $R[id_i]$ contains the view obtained by process id_i in this copy.
- An array $done[1, \ldots, N]$ of Boolean variables; all entries are initially **false**. Entry $done[id_i]$ indicates whether process id_i is done with this copy.
- A Boolean variable *allDone*, initially **false**. Indicates whether all processes which could be inside this copy are done.
- A Boolean variable *inside*, initially **false**. Indicates whether some process is already inside this copy.

With each copy, we associate a separate adaptive procedure for one-shot scan of the participating processes, latticeAgreement [7], with $O(k \log k)$ step complexity.

The pseudocode appears in Algorithm 1. For simplicity, we associate a virtual Boolean variable, *allDone*, with copy 0 of every sieve, whose value is **true**.

3.2 Properties of the Basic Sieve

As mentioned before, a process accessing the sieve gets a view of the processes in the sieve concurrently with it, or gets an empty view. Among the processes which get a view, there is a unique non-empty set of *candidates* which are seen by all processes; candidates are in the same copy of the sieve simultaneously. Some candidates are *winners*, who update this copy's data structures, e.g., increment *count* by 1, in a synchronized manner.

Algorithm 1 Long-lived adaptive sieve: code for process p_i.

data types:
  processID: int $1 \ldots N$ // process' id
  view: vector of $\langle ID:$ processID, $INFO:$ information field$\rangle$

view procedure enter(s, c: int, *info*: information field) // enter (s, c) with *info*
1: s.*inside*[c]=**true** // notify that there is a process inside copy (s, c)
2: V = s.latticeAgreement[c](*info*) // announce *info* (from [7])
3: $s.R[c][id_i] = V$ // save the obtained view
4: return candidates(s,c) // return the set of candidates

void procedure exit(s, c: int) // leave the sieve
1: $s.done[c][id_i] =$ **true** // p_i is done
2: W=candidates(s,c) // get the set of candidates
3: if $(id_i \in W)$ then $s.count = c + 1$ // p_i is a winner in (s, c)
4: if ($W \neq \emptyset$ and $\forall id_j \in W$, $s.done[c][id_j] ==$ **true**) then // candidates are done
5:   releaseSieve(s, c, W) // release the sieve

boolean procedure openFor(s, c) // check whether copy (s, c) is open
1: if ($s.allDone[c-1]$ and // all candidates of the previous copy are done
    not $s.inside[c]$) // and no process is inside the current copy
2: then return **all** // the copy is open for all processes
3: else return $\emptyset$ // the copy is open for no process

view procedure candidates(s,c) // returns the candidates of (s, c)
1: $V = s.R[c][id_i]$
2: $W = \min\{s.R[c][id_j] \mid id_j \in V$ and $s.R[c][id_j] \neq \emptyset\}$ // min by containment
3: if $\forall id_j \in W$, $s.R[c][id_j] \supseteq W$ then return W
4: else return $\emptyset$

void releaseSieve(s, c, W) // update data structures and release the sieve
1: $s.allDone[c] =$ **true** // release the sieve

For copy c of sieve s, procedure candidates(s, c) returns either an empty view (in Line 4) or a non-empty view (in Line 3). The key agreement property of the sieve is that all non-empty views returned by candidates(s, c) are equal:

Lemma 1. *If W_1 and W_2 are non-empty views returned by invocations of candidates(s, c) then $W_1 = W_2$.*

Process p_i is a *candidate* in copy c of sieve s if it appears in the non-empty view returned by candidates(c, s) (by Lemma 1, this view is unique). Process p_i is a *winner* in copy c of sieve s if the view obtained by p_i from candidates(s,c) contains p_i itself (see Line 2 of exit(s,c)), and in particular, is not empty. Note that a winner is in particular, a candidate.

Lemma 2. *If p_i is a candidate in copy c of sieve s, then p_i appears in every non-empty view returned by an invocation of* candidates*(s, c).*

Process p_i is *inside* copy c of sieve s when it sets $s.inside[c]$ to **true** in Line 1 of enter. A process inside copy c of sieve s is *done* after it assigns **true** to $s.done[c][id_i]$ in Line 1 of exit(s,c). The next lemma states that candidates are only inside a single copy of a specific sieve.

Lemma 3. *If process p_i is inside copy c of sieve s then all candidates of smaller copies, $1, \ldots, c-1$, of sieve s are done.*

By Lemmas 2 and 3 and the code, processes write the same values to $s.inside[c]$ and $s.allDone[c]$. Thus, each of these variables changes only once during the execution. Also, $s.count$ increases exactly by 1. This implies the next proposition:

Proposition 4 (Synchronization). *The variables of sieve s change in the following order (starting with $s.count = c-1$):*

1.	$inside[c-1] =$ ***true***	
2.	$count = c$	
3.	$allDone[c-1] =$ ***true***	(s, c) *is open*
4.	$inside[c] =$ ***true***	(s, c) *is not open*
5.	$count = c+1$	
6.	$allDone[c] =$ ***true***	$(s, c+1)$ *becomes open*
7.	$inside[c+1] =$ ***true***	$(s, c+1)$ *is not open*
8.	$count = c+2$ etc.	

It can be shown that processes enter a specific copy of a sieve *simultaneously*, and can not do so at different points. The next lemma states that at least one of the processes which are inside a copy of a sieve is a winner.

Lemma 5. *At least one of the processes which assign true to $s.inside[c]$ is a winner of copy c of sieve s.*

By Lemma 2, a non-empty view returned by candidates includes all candidates. It is possible that p_i obtains an empty view from candidates and is not a winner, yet later processes will see p_i in a non-empty view obtained from candidates.

Process p_i *accesses* copy (s, c) if it reads c from $s.count$; p_i *enters* (s, c) if it performs enter(s, c); otherwise, p_i *skips* (s, c).

The next lemma shows that a process does not win a sieve only if some concurrent process is a candidate in this sieve.

Lemma 6. *If p_i accesses sieve s and does not win, then there is a candidate of sieve s which is inside sieve s concurrently with p_i.*

The step complexity of enter is dominated by the $O(k \log k)$ step complexity of latticeAgreement [7]; the step complexity of exit is $O(k)$.

4 Polynomial Adaptive Long-lived $(2k-1)$-Renaming

Our algorithm employs a helping mechanism from the renaming algorithm of Gafni [17]. In Gafni's algorithm, a process tries to reserve names for processes with smaller priority and then obtains a name reserved for it. When there are several reservations for the same process, a reservation for a small name overrules reservations for larger names.

Gafni's algorithm is not long-lived as it has no mechanism for releasing a name or cancelling a reservation. His algorithm represents names with arrays; this is not adaptive since $O(N)$ operations are required to read the arrays.

Our algorithm extends the basic sieve (described in Section 3) to support *reservations*: If sieve s is reserved for process p_i, then only p_i can enter the sieve and get a new name s. Sieves can be entered repeatedly, in an adaptive manner, which makes them appropriate for long-lived adaptive algorithms.

As in the basic sieve, the modified sieve has a set of *candidates*; the sieve is reserved for the minimal process suggested by the candidates. In Gafni's algorithm, no reservation is made when several processes are suggested for the same name; in our algorithm, progress is made even if there is a collision since there is agreement on the reserved process.

In our algorithm, processes' priorities are based on *timestamps*. In a getName operation, process p_i acquires a timestamp TS_i, and the operation is identified by $\langle TS_i, id_i \rangle$. Timestamps are comparable, and earlier operations have smaller timestamps. In contrast, Gafni's algorithm uses static priorities for processes, based on their identifiers.

4.1 The Renaming Algorithm

In Algorithm 2, a process wishing to obtain a new name first gets a timestamp. Then, it repeatedly tries to reserve names for processes with smaller timestamps until it sees a reservation for itself. Finally, it gets a name from one of the sieves reserved for it.

To obtain the optimal name space, reservations are made only for processes which are active when getName starts. The set of active processes is maintained using the following adaptive procedures of Afek et al. [4]: In join(id_i), a process announces it is active, while in leave(id_i), a process announces it is no longer active; getSet returns the current set of active processes. It is guaranteed that processes which complete join before the beginning of getSet and start leave after the end of getSet appear in the set returned by getSet. A process completing join after the beginning of getSet or starting leave before the end of getSet may or may not appear in this set. The step complexity of these procedures is $O(k^3)$ [4].

Timestamps are obtained (in an adaptive manner) in procedure getTS, from a separate chain of sieves, using an idea we presented in [1]. A timestamp is a sequence of integers read from the counters of these sieves. Timestamps are compared by lexicographic order; if two timestamps have different lengths, then the shorter one is extended with the necessary number of $+\infty$. It can be shown that non-overlapping getTS operations return monotonically increasing timestamps.

In order to reserve a name, process p_i finds the smallest "empty" sieve which is not reserved, or which is overruled by a reservation in a smaller sieve. p_i enters the current copy of this sieve, announcing which process it is suggesting. Then, p_i computes the candidates of this copy; if all candidates of this copy are done, then p_i reserves the next copy of the sieve for the minimal process suggested by them, and exits the sieve immediately. Since processes agree on the suggestions of candidates, it is clear for which process to reserve the next copy.

To get a name for itself, p_i tries to enter sieves reserved for it. If p_i is the single winner in sieve s, then its new name is s; otherwise, p_i moves to the preceding (smaller) sieve; in this case, a smaller sieve must be reserved for p_i.

An unbounded array, $res[\langle TS_i, p_i \rangle]$, is used to notify process p_i that a reservation was made for its operation with timestamp TS_i. Section 6 discusses how to bound this array as well as the number of copies for each sieve.

4.2 The Modified Sieve

The basic sieve is modified so that if sieve s is *reserved* for an operation oid_i, then only process p_i performing oid_i can enter sieve s. The modified sieve supports the same operations as the basic sieve, with openFor extended as follows:

openFor(s, c): returns **all** if all operations can enter copy (s, c), $\emptyset$ if no process can enter copy (s, c), and oid_i if only oid_i can enter copy (s, c).

Algorithm 3 presents the modified procedures, openFor and releaseSieve.

Process p_i executing oid_i accesses the modified sieve in the same order as for the basic sieve, adapted to accommodate the change in openFor:

1. $c =$ read$(s.count)$ // find the current copy
2. if (openFor(s, c) == **all** or oid_i) // copy (s, c) is open or reserved for oid_i
3. then enter$(s, c, info)$ // enter (s, c) and announce $info$

...

4. exit(s, c) //leave the sieve

5 Proof of Correctness

5.1 Uniqueness

The uniqueness of new names follows from properties of the basic sieve, proved in Section 3.2. In getNameForSelf, a process gets a name s only if it is a single winner in sieve s (that is, candidates returns $\{oid_i\}$). Lemmas 1 and 3 imply:

Proposition 7. *No two processes hold the same name after a finite prefix.*

Algorithm 2 Long-lived $(2k-1)$-renaming: code for process p_i.

```
data types:
    timeStamp: string of integers
    operationID: ⟨TS : timeStamp, ID : processID⟩

int procedure getName( )        // get a new name from the range 1...2k − 1
    TS_i = getTS( )                                       // get a timestamp
    oid_i = ⟨TS_i, id_i⟩                          // id of the current operation
    currOID[id_i] = oid_i             // announce the id of the current operation
    join(oid_i)                                      // join the active set [4]
    O = getSet()                     // get operations of the active processes [4]
    repeat
        oid_r = min{oid ∈ O | res[oid] == ⊥}                  // try to reserve for
        reserve(oid_r)               // the earliest operation without a reservation
    until (start_sieve = res[oid_i] ≠ ⊥)                      // have a reservation
    return getNameForSelf(oid_i, start_sieve)  // get a name in a reserved sieve

void procedure reserve(oid_r)                             // try to reserve a sieve
    s = 0
    repeat forever                                              // loop over sieves
        s++
        c = s.count
        if (openFor(s, c) == all)    // copy is empty and without a reservation
            enter(s, c, oid_r )                           // try to reserve for oid_r
            exit(s, c)                                            // leave the sieve
            return

int procedure getNameForSelf( oid_i, start_sieve ) // get a name in a reserved sieve
    for (s = start_sieve down to 1)
        c = s.count
        if (openFor(s, c) == oid_i) then   // (s, c) is empty and reserved for oid_i
            W = enter(s, c, ⊥)                   // no suggestion for the next copy
            if (W == {oid_i}) then return s   // p_i is the single winner in sieve s
            else exit(s, c)                   // a smaller sieve is reserved for oid_i

void procedure releaseName( )
    exit(s, c) // leave the sieve; (s, c) is remembered from the last call to enter
    leave(oid_i)                                    // leave the active set [4]
```

Algorithm 3 The modified sieve: new code for openFor and releaseSieve.

data type:
accessList: operationID $\bigcup \{\emptyset, \textbf{all}\}$ // operations which can access a sieve

accessList procedure openFor(s, c) // which operations can access (s, c)
if (**not** $s.allDone[c-1]$ or $s.inside[c]$) then return $\emptyset$ // as in Algorithm 1
$oid_r = s.res[c]$
if ($oid_r == \bot$ // no reservation
  or for some $s' \in \{s-1, \ldots, 1\}$, $c' = s'.count$ and // reservation for oid_r is
    $s'.allDone[c'-1]$ and $s'.res[c'] == oid_r$ // overruled in a smaller sieve
  or $currOID[oid_r.ID] \neq oid_r$) // p_r started a new operation
then return **all** // the sieve is open for any operation
else return oid_r // the sieve is reserved for oid_r

void releaseSieve(s, c, W) // reserve (s, c) for the earliest operation suggested
$oid_r = \min_{\langle id,\, oid\rangle \in W} oid$ // earliest operation suggested by candidates
$s.res[c+1] = oid_r$ // reserve the next copy for oid_r
$s.allDone[c] = \textbf{true}$ // as in Algorithm 1
$res[oid_r] = s$ // notify p_r that it has a reservation in sieve s

5.2 Properties of the Reservations

Sieve s is *reserved* for operation oid_r after a finite prefix γ, denoted $Res(s, \gamma) = oid_r$, if and only if $s.allDone[s.count - 1] = \textbf{true}$ (candidates of the previous copy are done), $s.inside[s.count] = \textbf{false}$ (no process is in the current copy) and $s.res[s.count] = oid_r$ (the current copy is reserved for oid_r).

By Lemmas 2 and 3 and the code, winners of copy (s, c) reserve the next copy for the same operation. This implies following extension of Proposition 4:

Proposition 8 (Synchronization). *The variables of a sieve s change in the following order (starting with $s.count = c - 1$):*

1. $inside[c-1] = \textbf{true}$
2. $count = c$
3. $res[c] = oid_r$
4. $allDone[c-1] = \textbf{true}$ — *(s, c) becomes reserved for oid_r*
5. $inside[c] = \textbf{true}$ — *(s, c) is no longer reserved*
6. $count = c + 1$
7. $res[c+1] = oid_{r'}$
8. $allDone[c] = \textbf{true}$ — *$(s, c+1)$ becomes reserved for $oid_{r'}$*
9. $inside[c+1] = \textbf{true}$ — *$(s, c+1)$ is no longer reserved*
10. $count = c + 2$ etc.

The next lemma shows that if openFor(s, c) returns oid_i, then sieve s is indeed open for oid_i. The first three statements mean that $Res(s, \gamma) = oid_i$.

Lemma 9. *If* openFor(s, c) *returns* oid_r*, then there is a prefix* γ *which ends on the execution interval of Line 1 of* openFor(s, c)*, such that* $s.allDone[s.count - 1] = \textbf{\textit{true}}$*,* $s.inside[s.count] = \textbf{\textit{false}}$*,* $s.res[s.count] = oid_r$*, and* $s.count = c$ *at the end of* γ*.*

$MinRes(oid_i, \gamma)$ denotes the minimal sieve reserved for an operation oid_i (of process p_i) after an execution prefix γ. Since a reservation for oid_i in a smaller sieve overrules reservations for this operation in larger sieves, $MinRes(oid_i, \gamma)$ is the actual reservation for oid_i. For convenience, it is ∞ if no sieve is reserved for oid_i. It is 0 after p_i enters the copy of the sieve from which it gets its new name; i.e., p_i writes **true** into $s.inside[s.count]$, where $s.count$ is the copy of sieve s from which p_i gets its new name.

We can prove that if openFor(s, c) returns **all**, then either there is no reservation in sieve s or the reservation is overruled by a smaller sieve. This allows to show that a process does not destroy the minimal reservation for another process. This is the key to proving that $MinRes$ monotonically decreases:

Lemma 10. *Assume* γ *is a finite prefix. Then for every prefix* γ' *of* γ *and every operation* oid_i*,* $MinRes(oid_i, \gamma') \geq MinRes(oid_i, \gamma)$*.*

The next lemma shows that openFor recognizes the minimal reservation.

Lemma 11. *If* openFor(s, c) *returns* oid_r*, then there is a prefix* γ *which ends on the execution interval of* openFor(s, c)*, such that* $MinRes(oid_r, \gamma) = s$*.*

Proof. By Lemma 9 there is a prefix γ which ends on the execution interval of Line 1 of openFor such that $Res(s, \gamma) = oid_r$. Therefore, $MinRes(oid_r, \gamma) \leq s$.

Assume, by way of contradiction, that $MinRes(oid_r, \gamma) = s' < s$. Let γ' be the shortest prefix such that $MinRes(oid_r, \gamma') = s'$. Suppose that p_i reads $s-1.count, \ldots, 1.count$ (in Line 4 of openFor) at the ends of prefixes $\gamma_{s-1}, \ldots, \gamma_1$, respectively. By the code, $\gamma_{s-1}, \ldots, \gamma_1$ end after the end of γ.

If $\gamma_{s'}$ is included in γ', then by Lemma 10, $s > MinRes(oid_r, \gamma_{s-1}) \geq \ldots \geq MinRes(oid_r, \gamma_{s'}) \geq s'$. By the pigeon hole principle, $MinRes(oid_r, \gamma_\ell) = \ell$ for some $\ell \in \{s-1, \ldots, s'\}$. By the sieve's properties, p_i reads $\ell.allDone[c-1] =$ **true**, and $\ell.res[c] = oid_r$ (since these variables do not change after γ_ℓ). Thus, the condition in Line 4 of openFor holds for p_i and openFor returns **all**. This is a contradiction, and the lemma follows.

Otherwise, $\gamma_{s'}$ is not included in γ'. Since s' is the minimal sieve reserved for oid_r, no process writes to $s'.count$ from the end of γ' until p_r writes a new value to $currOID[oid_r]$. Since p_i reads $s'.count$ before p_r writes a new value to $currOID[oid_r]$, the sieve's properties imply that p_i reads **true** from $s'.allDone[c-1]$, and oid_r from $s'.res[c]$ (since these variables do not change after $\gamma_{s'}$). Thus, the condition in Line 4 of openFor holds for p_i and openFor returns **all**. This is a contradiction, and the lemma follows. □

5.3 Size of the Name Space

Let δ_i be an interval of getName$_i$; p_i has several attempts (calls to reserve) to make a reservation; let β_i be the interval of one invocation of reserve. Partition β_i into disjoint intervals during which p_i accesses different sieves: If p_i accesses sieve s then $\beta_i|_s$ is the interval which starts when p_i reads from $s.count$ (or the beginning of β_i, if $s = 1$) and ends when p_i reads from $s + 1.count$ (or the end of β_i, if p_i is a winner in sieve s).

The next lemma shows that when a process skips a sieve in reserve, either some concurrent process is a candidate in this sieve or the sieve is reserved for another process. It follows from Lemmas 6 and 11 and the code.

Lemma 12. *If in reserve p_i skips sieve (s, c) in reserve then either*
(1) there is a candidate in (s, c) concurrently with p_i, or
(2) there is an operation oid_r such that $MinRes(oid_r, \gamma) = s$, for some prefix γ ending in $\beta|_s$.

A potential method is used to show that only sieves $< 2k$ are reserved; this bounds the name space, since a process obtains its new name only from a sieve reserved for it. We define two sets of simultaneously active operations and show that at least one of them grows by 1 when p_i skips a sieve in reserve. Each set contains at most $k - 1$ concurrent operations (except p_i); this implies p_i skips at most $2k - 1$ sieves, and makes a reservation in a sieve $\leq 2k$. However, as we show, the *minimal* reservation for a process is always $< 2k$.

The first set, E_s, contains operations whose attempt ends in a sieve $\leq s$, while the second set, R_s, contains operations reserved in a sieve $\leq s$. Formally, for a prefix γ and a process p_i which performs an attempt with interval β_i, define:

$$E_s(\gamma) = \{oid_q \mid q \text{ is a candidate in a sieve} \leq s \text{ whose attempt covers the end of } \gamma\}$$

$$R_s(\gamma, p_i) = \{oid_q \mid q \text{ is active at the end of } \gamma \text{ and } MinRes(oid_q, \beta_i|_s) \leq s\}$$

E contains operations whose *attempt* covers the end of γ while R contains operations whose *whole* execution interval covers the end of γ. This reflects the fact that E depends on an operation's execution, while R depends on the minimal sieve reserved for an operation by others. In the definition of R_s, γ is used to determine which operations are active, but $MinRes$ is evaluated at the end $\beta_i|_s$.

The next key lemma shows that a process accesses a sieve with a high index only if many processes are active concurrently with it. The proof is by induction on the sieve's number, and considers the different cases in which a process skips a sieve; the proof is omitted due to lack of space.

Lemma 13. *Let β_i be the interval of an attempt by p_i. If p_i skips sieve s, then there is a prefix γ which ends in $\beta_i|_1\beta_i|_2 \ldots \beta_i|_s$ such that $|E_s(\gamma)| + |R_s(\gamma, p_i)| \geq s$.*

The next lemma shows that the minimal sieve reserved for a process is $< 2k$.

Lemma 14. *Let δ_i be the execution interval of oid_i of p_i If for some prefix γ_i, $MinRes(oid_i, \gamma_i)$ is not ∞, then $MinRes(oid_i, \gamma_i) < 2k$, where $k = PntCont(\delta_i)$.*

Proof. Let $\mathit{MinRes}(oid_i, \gamma_i) = s$, and let p_j be a process which suggests oid_i in sieve s. Let $\beta_j|_s$ be the interval of p_j's attempt in which it skips sieves $1, \ldots, s-1$ and suggests oid_i in $(s, c-1)$. By Lemma 13, $s-1 \leq |R_{s-1}(\gamma_j, p_j)| + |E_{s-1}(\gamma_j)|$, for some prefix γ_j which ends in $\beta_j|_s$. Let $k = \mathrm{PntCont}(\beta_j|_s)$; $k \leq \mathrm{PntCont}(\delta_i)$, since $\beta_j|_s$ is contained in δ_i. By definition, $R_{s-1}(\gamma_j, p_j)$ and $E_{s-1}(\gamma_j)$ contain operations which are simultaneously active at the end of γ_j; hence, $|R_{s-1}(\gamma_j, p_j)| \leq k-1$. By definition, $oid_j \notin E_{s-1}(\gamma_j)$, which implies that $|E_{s-1}(\gamma_j)| \leq k-1$.

If $oid_i \in R_{s-1}(\gamma_j, p_j)$, then by the definition of R_{s-1}, $\mathit{MinRes}(oid_i, \beta_j|_{s-1}) \leq s-1$. The properties of the sieve imply that p_j accesses $(s, c-1)$ before sieve s is reserved for oid_i. That is, $\beta_j|_{s-1}$ ends before γ_i. Therefore, by Lemma 10, $\mathit{MinRes}(oid_i, \gamma_i) \leq \mathit{MinRes}(oid_i, \beta_j|_{s-1}) \leq s-1$. The lemma follows since $s-1 \leq |R_{s-1}(\gamma_j, p_j)| + |E_{s-1}(\gamma_j)| \leq 2k-1$.

If $oid_i \notin R_{s-1}(\gamma_j, p_j)$, then $|R_{s-1}(\gamma_j, p_j)| \leq k-1$. In this case, the lemma follows since $s \leq |R_{s-1}(\gamma_j, p_j)| + |E_{s-1}(\gamma_j)| + 1 \leq 2k-1$. □

This proves that the minimal reservation for an operation is in a sieve $< 2k$. We now show that p_i succeeds to get a name from one of the sieves reserved for it. The next lemma states that if p_i skips sieve s in getNameForSelf, then a smaller sieve is reserved for p_i.

Lemma 15. *Assume that in getNameForSelf, p_i reads $s.count$ at the end of prefix γ_s and $(s-1).count$ at the end of prefix γ_{s-1}; that is, p_i skips sieve s. If $\mathit{MinRes}(oid_i, \gamma_s) \leq s$ then $\mathit{MinRes}(oid_i, \gamma_{s-1}) \leq s-1$.*

Finally, we prove that the algorithm provides optimal name space.

Lemma 16. *getNameForSelf returns a new name in the range $\{1, \ldots, 2k-1\}$.*

Proof. By the algorithm, p_i calls getNameForSelf (Line 8 of getName) after it reads a non-$\perp$ value, say, ss, from $res[oid_i]$. Sieve ss is reserved for oid_i before ss is written to $res[oid_i]$.

Let s_m be the minimal value of $\mathit{MinRes}(oid_i)$. In getNameForSelf, p_i accesses sieves $ss, ss-1, \ldots$. If p_i does not win some sieve s, then Lemma 15 implies that a smaller sieve is reserved for p_i. Thus, p_i wins sieve s_m at latest and getNameForSelf returns a name $s > 0$.

It remains to show that the new name is $< 2k$. By the algorithm, openFor(s, c) returns oid_i (Line 3). Lemma 11 implies that $\mathit{MinRes}(oid_i, \gamma) = s$, for some prefix γ. By Lemma 14, $\mathit{MinRes}(oid_i, \gamma) < 2k$, and hence, $s < 2k$. □

5.4 Step Complexity

Let *elected*(s, c) be the minimal operation identifier suggested by candidates in copy (s, c). By Lemma 1, processes agree on the set of candidates and the next copy of the sieve $(s, c+1)$ is reserved for *elected*(s, c). If process p_i gets its new name from copy (s, c), then p_i is the single candidate in (s, c) and it writes $\perp$ in Line 5 of exit; in this case, *elected*$(s, c) = \perp$. Since a process makes suggestions in one sieve at a time, we can bound the number of different copies in which the same operation is elected.

Lemma 17. *An operation is elected in at most $2k$ copies.*

Line 6 of getName guarantees that p_i repeatedly calls reserve as long as no reservation was made for it. To bound the number of attempts p_i makes, we bound the number of different operations encountered by p_i as $elected(s, c)$ and the number of copies they are elected in. Using similar arguments, we also prove that $O(k)$ attempts of p_i end in a copy (s, c) such that $elected(s, c) = \perp$.

Lemma 18. *$O(k^2)$ attempts of p_i end in a copy (s, c) such that $elected(s, c) = oid_j \neq \perp$. $O(k)$ attempts of p_i end in a copy (s, c) such that $elected(s, c) = \perp$.*

Thus, p_i discovers that a sieve is reserved for oid_i and calls getNameForSelf after $O(k^2)$ attempts. In each attempt, p_i skips $O(k)$ sieves and enters one sieve; in getNameForSelf, p_i enters $O(k)$ sieves. Skipping a sieve requires $O(k)$ steps (Line 9 of reserve); entering a sieve requires $O(k \log k)$ steps [7]. Therefore, the total step complexity of the algorithm is $O(k^2[k \cdot k + k \log k] + k \cdot k \log k) = O(k^4)$.

6 Discussion

We present a long-lived $(2k-1)$-renaming algorithm with $O(k^4)$ step complexity, where k is the maximal number of simultaneously active processes in some point of the execution.

The algorithm described here uses an unbounded amount of shared memory. First, each sieve has an unbounded number of copies. Copies can be re-cycled (and hence bounded), see [1]. Second, an unbounded array is used to notify a process about reservations for each of its operations. This array can be replaced by a two-dimensional array in which each process announces for each other process the last timestamp it reserved for (and in which sieve). A process checks, for every process in the current active set whether a reservation was made for its current timestamp; then it tries to enter from the minimal reservation made for it, if there are any. The details are postponed to the full version of the paper.

References

1. Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Adaptive long-lived renaming using bounded memory. Available at www.cs.technion.ac.il/~hagit/pubs/AAFST99disc.ps.gz, Apr. 1999.
2. Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-lived renaming made adaptive. In *Proc. 18th ACM Symp. Principles of Dist. Comp.*, pages 91–103, 1999.
3. Y. Afek and M. Merritt. Fast, wait-free $(2k-1)$-renaming. In *Proc. 18th ACM Symp. Principles of Dist. Comp.*, pages 105–112, 1999.
4. Y. Afek, G. Stupp, and D. Touitou. Long-lived and adaptive collect with applications. In *Proc. 40th IEEE Symp. Foundations of Comp. Sci.*, pages 262–272, 1999.

5. Y. Afek, G. Stupp, and D. Touitou. Long-lived and adaptive atomic snapshot and immediate snapshot. In *Proc. 19th ACM Symp. Principles of Dist. Comp.*, 2000. To appear.
6. H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–548, July 1990.
7. H. Attiya and A. Fouren. Adaptive wait-free algorithms for lattice agreement and renaming. In *Proc. 17th ACM Symp. Principles of Dist. Comp.*, pages 277–286, 1998.
8. H. Attiya and A. Fouren. An adaptive collect algorithm with applications. Submitted for publication. Available at www.cs.technion.ac.il/~hagit/pubs/AF99ful.ps.gz, Aug. 1999.
9. H. Attiya and A. Fouren. Adaptive long-lived renaming with read and write operations. Technical Report 956, Faculty of Computer Science, The Technion, Mar. 1999. Available at www.cs.technion.ac.il/~hagit/pubs/tr0956.ps.gz.
10. A. Bar-Noy and D. Dolev. A partial equivalence between shared-memory and message-passing in an asynchronous fail-stop distributed environment. *Math. Sys. Theory*, 26(1):21–39, 1993.
11. E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proc. 25th ACM Symp. Theory of Comp.*, pages 91–100, 1993.
12. E. Borowsky and E. Gafni. Immediate atomic snapshots and fast renaming. In *Proc. 12th ACM Symp. Principles of Dist. Comp.*, pages 41–52, 1993.
13. E. Borowsky, E. Gafni, N. Lynch, and S. Rajsbaum. The BG distributed simulation algorithm. Technical Report MIT/LCS/TM-573, Laboratory for Computer Science, MIT, Dec. 1997.
14. H. Buhrman, J. A. Garay, J.-H. Hoepman, and M. Moir. Long-lived renaming made fast. In *Proc. 14th ACM Symp. Principles of Dist. Comp.*, pages 194–203, 1995.
15. J. E. Burns and G. L. Peterson. The ambiguity of choosing. In *Proc. 8th ACM Symp. Principles of Dist. Comp.*, pages 145–158, 1989.
16. A. Fouren. Exponential examples for two renaming algorithms. Available at www.cs.technion.ac.il/~hagit/pubs/expo.ps.gz, Aug. 1999.
17. E. Gafni. More about renaming: Fast algorithm and reduction to the k-set test-and-set problem. Unpublished manuscript, 1992.
18. M. Herlihy and N. Shavit. A simple constructive computability theorem for wait-free computation. In *Proc. 26th ACM Symp. Theory of Comp.*, pages 243–252, 1994.
19. L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, Feb. 1987.
20. M. Moir. Fast, long-lived renaming improved and simplified. *Sci. Comput. Programming*, 30(3):287–308, May 1998.
21. M. Moir and J. H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Programming*, 25(1):1–39, Oct. 1995.
22. M. Moir and J. A. Garay. Fast long-lived renaming improved and simplified. In *Proc. 10th Int. Workshop on Dist. Algorithms*, volume 1151 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 1996.

This article was processed using the LaTeX macro package with LLNCS style