Fast, wait-free $(2k-1)$-Renaming

(Extended abstract)

Yehuda Afek* Michael Merritt†

Abstract

We describe a fast, wait-free $(2k-1)$-renaming algorithm which takes $O(k^2)$ time. (Where k is the contention, the number of processes actually taking steps in a given run.) The algorithm makes extensive use of tools and techniques developed by Attiya and Fouren [AF98]. Other extensions, including a fast (long-lived) atomic snapshot algorithm, are briefly discussed.

1 Introduction

Since early work in mutual exclusion [Lam87], researchers have asked whether distributed algorithms can be made *fast*; that is, can their worst-case time complexity be bounded by a function of the number of actually active or contending processes, rather than the total number of processes that *might* take steps [ADT95][1]? If a fast solution can be found for a given problem, is there a trade-off in other measures?

Attiya and Fouren examined several problems and provide several fast constructions for wait-free lattice agreement[2] and the wait-free renaming problem[3] [AF98]. Among their fast solutions to the renaming problem, the one with the smallest output name space reduces the name space to $6k-1$ names, where k is the number of active processes. This leaves open the question whether a sub-optimal name space is an inevitable price of utliyzing a fast algorithm. Our work closes the remaining gap, demonstrating that an optimal name space of $2k-1$ can be achieved with a fast algorithm.

Specifically, we describe an algorithm for reducing names from a space of size $f(k)$ to $2k-1$, in time $O(f(k)^2)$, where $f(k)$ is a bound on the largest input name among any k active processes. If we use this algorithm after running the fast Attiya-Fouren algorithm for $6k-1$-renaming in time $O(k \log k)$ [AF98], so that $f(k) = 6k-1$, the result is a fast $(2k-1)$-renaming algorithm which takes $O(k^2)$ time.

Our algorithm is a modification of another algorithm by Attiya and Fouren, which solves $(2k-1)$-renaming but has $O(N)$ worst case time complexity [AF98], where N is a bound on the original name space. This worst-case complexity arises even in runs in which the largest input name of an active process is very small relative to N.

Our modification of Algorithm 2 in [AF98] remains optimal in the size of the output name space, and reduces the time complexity to a function of the largest input name of an active process. Yet, the original input name space is not necessarily sensitive to the number of active processes–the input name of the only active process might be N. To this end, a fast renaming algorithm can be run first, which reduces the input name space to some function of the number of active processes, $f(k)$, so that the largest output name of this algorithm, bounded by $f(k)$, *is* sensitive to the number of active processes. (Candidate algorithms include Attiya and Fouren's adaptation [AF98] of Moir and Anderson's k^2-renaming algorithm [MA95], or Attiya and Fouren's $6k-1$-renaming algorithm which have worst-case time complexity of k and $k \log k$, respectively.) Hence, if our algorithm is run subsequently, with these names as inputs, the result is a fast $2k-1$-renaming algorithm. (The resulting composite algorithms have worst-case time complexities of $O(k^4)$ and $O(k^2)$, respectively.)

*Tel Aviv University, Tel-Aviv, Israel. afek@math.tau.ac.il. Part of this work was performed while visiting at AT&T Labs.

†AT&T Labs, 180 Park Av., Florham Park, NJ 07932-0971. mischu@research.att.com.

[1]Others describe algorithms satisfying this property *adaptive*, e.g. [AF98].

[2]Lattice agreement is a linearizable decision-problem in which each process *writes* its name, and then performs a *snapshot* operation, which returns the subset of processes which have previously written.

[3]In the f-renaming problem, as many as n processes execute *rename* operations. They begin with unique names taken from a set of size N, and exit with unique names from a set of size $f < N$. For wait-free algorithms utilyzing read-write registers, f must be at least $2n-1$ [HS93]. Algorithms which are not fast meet this bound [ABND+90].

PODC '99 Atlanta GA USA
Copyright ACM 1999 1-58113-099-6/99/05...$5.00

2 Algorithm and analysis

Although the algorithm below and the correctness proofs in the next section are self-contained, the discussion below assumes familiarity with the work of Attiya and Fouren [AF98].

2.1 Algorithm overview

The task is to reduce a namespace from $f(k)$ names to $2k-1$. To do this, we modify the Attiya-Fouren $(2k-1)$-renaming algorithm [AF98], which has $O(N)$ worst case time complexity, and show how to make it fast. As described in Figure 1, their algorithm depends on a two-dimensional array of *reflectors*, each consisting of two binary registers. Each process enters the array in the column whose index corresponds to its input name, moves down through the column until it enters a reflector that has been set by another process moving horizontally to the right, and then turns to the right and moves horizontally through the remaining columns. (If no reflector has been set by other processes, it turns and starts moving horizontally to the right at the bottom reflector in the next column.) The row at which it exits is its output name. This array is of width N and height at most $2N-1$.

We size this array by setting $N = f(n)$, where $n \leq N$ is an *a priori* upper bound on k, the number of active processes in a given run. Our first observation is that with the array sized to $f(n)$, by labeling the columns of this array from right to left, so that low-numbered processes enter columns near the right edge of the array, the array is only accessed in the rightmost $f(k)$ columns.

In the Attiya-Fouren algorithm, processes move vertically through their input column until colliding with a horizontally-moving process or hitting the bottom of the array, and then turn and move horizontally to the right. A property of this algorithm is that no process moves horizontally in any row above the $2k-1$ bottom-most rows. (No process turns from vertical to horizontal unless it hits the bottom of the array or another horizontally-moving process.) Hence, any reflector above the bottom $2n-1$ rows is only traversed vertically and is superfluous. Hence, our second observation

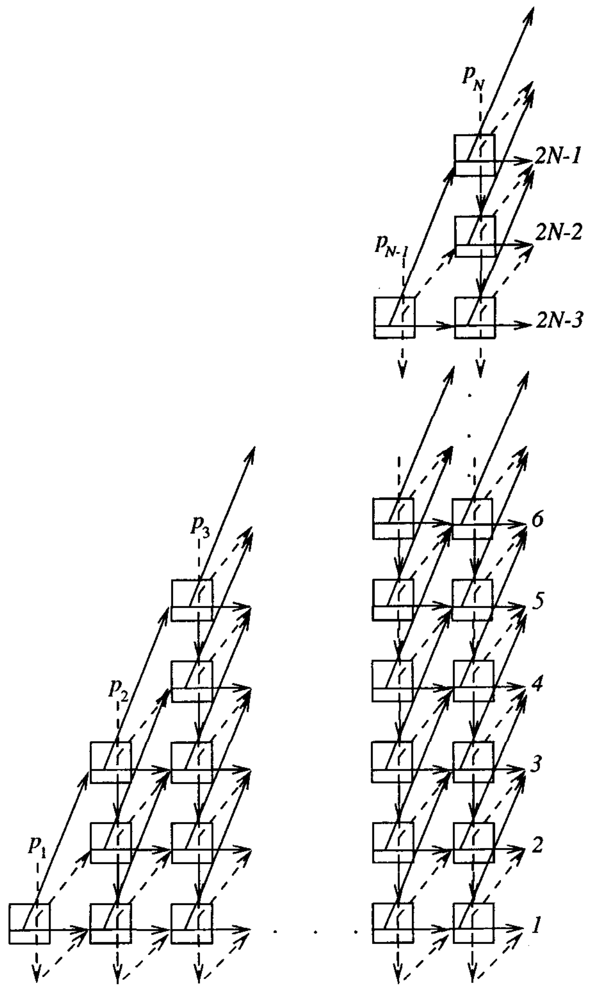

Figure 1: Reflector array for $2k-1$-renaming in $O(N)$ time [AF98].

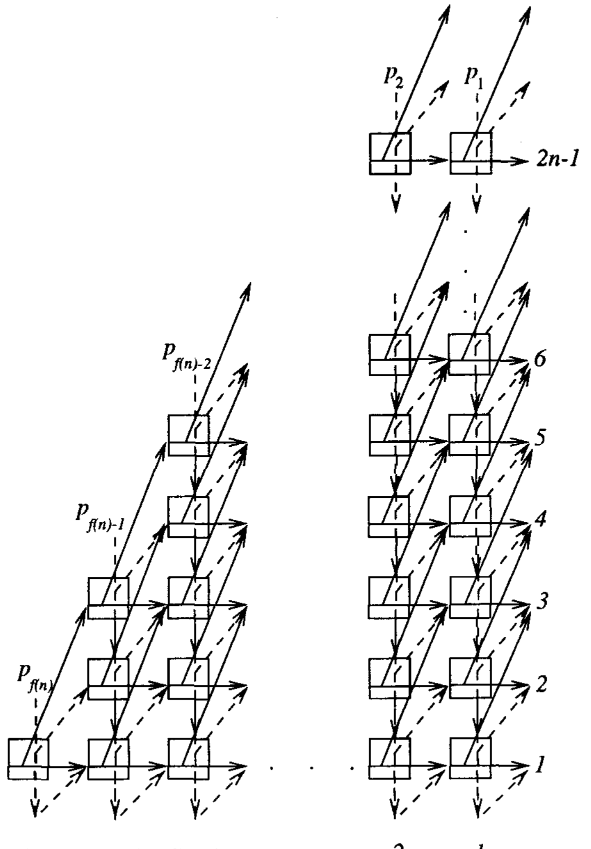

Figure 2: Modified reflector array for $2k-1$-renaming in $O(n + f(k))$ time.

is that we can remove all but the bottom $2n-1$ rows (of the $2N-1$ original rows) of the array without changing the correctness of the algorithm. (See Figure 2.) These observations together change the time complexity from $O(N)$ to $O(n+f(k))$.

The remaining problem is to change this time complexity from $O(n+f(k))$ to $O(f(k)^2)$. As explained in the next section, this is achieved by restricting the processes to be active in the bottom $2f(k)$ rows of the array, at the cost of up to $2f(k)$ steps in *each* column by *each* process. Hence, instead of accessing n reflectors in its entering column and one in each of as many as $f(k)$ other columns, each process may access as many as $2f(k)$ reflectors in each of $f(k)$ columns.

As in the Attiya-Fouren algorithm, processes move through the array from left to right, starting from their input column (indexed by their input name) down to the first column. But in each column to the right of their input column, instead of accessing the single reflector at their current index, they either pass through the reflectors from the bottom row up to their current index, or from an estimate of the top row down toward the bottom. They run the fast snap-shot (lattice-agreement) algorithm of Attiya and Fouren [AF98] to determine an adequate estimate of the top row that is no bigger than $2f(k)$. Race conditions introduce some ambiguity, but in any run, all processes moving vertically in a column (in either direction) will stop and turn right at one of two neighboring reflectors.

2.2 Code and explanation

As outlined in the previous section, the algorithm in Figure 3 can be understood as an adaptation of the Attiya-Fouren algorithm [AF98] for $2k-1$-renaming in $O(N)$ time. We described two simple modifications of that algorithm that reduce the time complexity to $O(n+f(k))$. In this bound, the $O(n)$ term arises because process p enters the array at column p, row $2n-1$, and passes down through the column, possibly as far as row 1. But since the other active processes are restricted to the bottom $2k-1$ rows, if p had an estimate for k, it could use that estimate to enter the array above them, but well below the $2n-1$ bound.

Here another building block from Attiya and Fouren's work is useful: using the fast lattice-agreement algorithm of Attiya and Fouren [AF98], process p performs a *write* and then a one-shot *snapshot* operation, returning a set $snap_p = \{s_m, ..., s_1\}$ of active processes, where $s_m > ... > s_1$. In this way p learns about other active processes, and is assured that any active processes not in $snap_p$, will return p in their own snapshots. (Again, every active process is either in p's snapshot, or sees p in its own snapshot. Of course, a process may be in p's snapshot and also see p in its own snapshot.) We show (Lemma 3) that p can use the largest input name in

```
    reflector:struct of bits up and down, initially false
    column[1 : 2n]: array of reflectors

shared variables:
    network[1 : f(n)]: array of columns
        /* (Reflectors above network[i][2f(n) - 2i + 1]
                                    are unnecessary.) */
local variables:
    snap: set of input names
    c: column
    s, col, maximum, index, j, last : integers

procedure rename(p)
/* Returns new name for process with input name p. */
    write(p)
                        /* Record p in snapshot object, */
    snap := snapshot
/* using Attiya-Fouren $O(k \log k)$ lattice agreement. */
    s := $max_{e \in snap}$ e
    for col = p down to 1 do
/* For each column from network[p] to network[1]. */
        if col ∈ snap then top(col)
        else bottom(col)
        if col = p then
                        /* First column for this processor. */
           index := $1 + \max\{\{0\} \cup \{j | j \le 2s - 2p - 1$
                              and network[p][j].down$\}\}$
        else if network[col][index].up
           then index := index + 2
    od
    return(index)

procedure top(c)
  /* Set the up bits in column c from the 2s - 2c - 1st
                                          down to 1, */
    for j = max(1, 2s - 2c - 1) down to 1 do
/* as long as the corresponding down bit is not set. */
        if network[c][j].down then return
        network[c][j].up := true
        if network[c][j].down then return
    od

procedure bottom(c)
 /* Set the down bits in column c from 1st to index */
    for j = 1 to index do
    /* as long as the corresponding up bit is not set. */
        if network[c][j].up then return
        network[c][j].down := true
        if network[c][j].up then return
    od
```

Figure 3: Code for process with input name p.

its snapshot, s_m, enter column p at row $2s_m - 2p - 1$, and be assured this reflector is at least as high as that accessed by any process in $snap_p$.

But what about the active processes that are not in $snap_p$? There may be many of them and they may access reflectors well above $2s_m - 2p - 1$. But they will in turn see p in their own snapshot. Knowing that p is active, they take p's role and also enter the algorithm from above, moving down vertically through the column.

Multiple processes moving vertically through a column requires other adjustments to the Attiya and Fouren algorithm: an initial read is necessary in each reflector access, to assure later processes actions are consistent with earlier. In addition, a process not observing p in its snapshot and moving horizontally to the reflector at row i in column p, will first access the reflectors below row i, starting at row 1 and moving up as far as row i or until meeting a process moving down through the column.

2.2.1 Estimating an entry point in a column

The processes moving from a high point down in a column need to know a place to start that is as high as the top-most reflector reached by any process moving from the bottom in that column, but the index of this position must be a function of k, not e.g. $2n$, as in the Attiya-Fouren algorithm. Let c be a column, let $snap_{max}$ be the largest snapshot returned that does not include c, and let s_{max} be the largest input name in $snap_{max}$. We claim that only processes in $snap_{max}$ (and among them, those which have input name greater than c) enter column c from the bottom, and the highest index they might reach in c (setting the down bit to true) is $2(s_{max} - c) + 1$. (Lemma 5.) Although s_{max} is not known to the processes, if a process observes c in its own snapshot, $snap$, (and hence enters column c from the top), then $snap_{max} \subset snap$, and so the maximum name in $snap$, s, is at least as big as s_{max}. Hence, entering column c at reflector $2s - 2c + 1$ is guaranteed to be high enough to be above any reflector reached by any of the processes (members of $snap_{max}$) moving from the bottom in column c. (Lemma 4.) Since $2s \leq 2f(k)$, $2s - 2c + 1$ provides the required "safe" entry point to the column.

2.2.2 Summing up

Another key argument is a case analysis that shows that no two process enter a column with the same index. (Lemmas 7 and 8.)

To see the time bound of $O(f(k)^2)$, consider that each process enters the array at column $c \leq f(k)$, at height $s \leq 2f(k)$, then passes through each lower-numbered column, either from a height at most $2f(k)$ down, or from the bottom to a height at most $2f(k) - 1$, taking a constant number of steps at each reflector, for a total of $O(f(k)^2)$ steps in the reflector array. The *write* and *snapshot* operations can be carried out in $O(k \log k)$ time using the Attiya-Fouren Lattice Agreement algorithm. Since $f(k) \geq k$, the result follows.

Theorem 1 *The algorithm in Figure 3 is a $(2k-1)$-renaming algorithm with $O(f(k)^2)$ step complexity.*

Proof: Lemma 2 implies the biggest name assigned is $2k - 1$, and Lemma 9 implies all names assigned are unique. Each process performs an order $k \log k$ snapshot algorithm, and then enters at most $f(k)$ columns, either visiting at most $2k - 1$ reflectors, from the bottom up (by Lemma 2), or $2f(k)$ reflectors from the top down, taking at most a constant number of steps in each. ■

Corollary 1 *There is a $(2k-1)$-renaming algorithm with $O(k^2)$ step complexity.*

Proof: Processes first execute the $(6k - 1)$-renaming algorithm of Attiya and Fouren [AF98], then execute the algorithm above with $f(k) = 6k - 1$. ■

3 Fast atomic snapshots and other fast building blocks

In another interesting extension of previous work in fast algorithms, one can apply ideas from Afek, *et al.* [AADGMS] to extend the fast *collect* algorithm of Attiya and Fouren [AF98] to produce the stronger semantics of a (long-lived), fast *atomic snapshot* primitive. This primitive provides an abstraction of a single-writer, multi-reader shared memory in which each address may be (repeatedly) written by a single process, and read by all others. In addition, it supports a *scan* operation, which returns in an atomic operation the values of all memory location which were written. The complexity of all operations in the fast algorithm are $O(k^2)$, where as usual k is the number of processes taking steps (details in the full paper). This fast snapshot primitive can in turn be used modularly in other constructions–for example, we are actively investigating the implications for constructing fast concurrent timestamp systems [GLS95] or fast randomized consensus [AH90]

The work of Moir and Anderson [MA95] and of Attiya and Fouren [AF98] provide an interesting and powerful set of fast algorithms and techniques which can be used modularly to construct fast solutions to other fault-tolerant distributed computing problems in the read-write register model. For example, the $O(n^2)$, $(2k-1)$-renaming algorithm in this paper uses Attiya and Fouren's lattice-agreement and $6k - 1$ algorithm as components. Other modular constructions and extensions of their work are also possible, and suggest that many other problems have fast solutions with low worst-case step complexity.

For example, subsequent to the work on fast renaming described here, Gafni observed that the fast, $O(k)$-step complexity *collect* procedure of Attiya and Fouren can be used modularly with previously known algorithms for $(2k-1)$-renaming to create fast solutions [G98]. Examples of such algorithms include Gafni's $O(n^3)$ algorithm [G92], Bar-Noy and Dolev's $O(n4^n)$ solution[BD89], and the algorithm of Burns and Peterson [BP89], which is cited as having $O(n^3)$ step complexity by Moir [M]. Use of the fast collect primitive with these algorithms results in fast, wait-free $(2k-1)$-renaming algorithms with worst-case step complexity of $O(k^3)$, $O(n4^n)$, and $O(k^3)$, respectively.

Other extensions lead to still more useful, fast building blocks. For example, the $(6k-1)$-renaming algorithm of Attiya and Fouren can be easily modified to use fewer names, to a limit of $4k-1$, without affecting the asymptotic worst-case step complexity. (The original algorithm is built modularly from components sized as increasing powers of 2. The number of names is reduced if instead the components are sized as powers of 4, and still farther if powers of 8 are used, with $4k-1$ names used in the limit as the base increases.)

4 Detailed proofs

We focus on *complete executions* of the algorithm: finite runs in which every process that carries out a rename operation completes it. Since the algorithm has no unbounded loops and makes calls to registers or to wait-free procedures (to *write*(p) and *snapshot*), each call will return if the calling process takes steps: the algorithm is wait-free. For the safety properties we consider below, we need consider only finite executions, each of which is a prefix of some finite complete execution.

For notational convenience, we identify processes with their unique input name between $f(k)$ and 1.

Lemma 1 *In any complete execution α,*

- *if process p has index j exiting column $c+1$, then it has index at most $j+2$ exiting column c.*
- *If process p's index j is the maximum index computed by any process exiting column $c+1$, then the process with input name c has index at most $j+1$ exiting column c.*

Proof: The first claim is a simple observation on the code for updating an index (which either leaves the index unchanged or increases it by two in each column). The second claim is a consequence of the code for setting indices for the process with name c, which is one more than the maximum i such that *network*$[c][i]$.*down* = true, which is at most j. ■

Following Attiya and Fouren [AF98], the next lemma shows that the output name space is at most $2k-1$, where k is the number of processes which execute renaming operations.

Lemma 2 (AF98) *Let α be a complete execution in which only the processes with input names $s_k, \ldots, s_1$ perform renaming operations, where $s_k > \ldots > s_1$. Then for all j, $k \geq j \geq 1$, processes s_k through s_j exit column s_j and any columns between s_j and s_{j-1} with indices no greater than $2k-2j+1$.*

Proof: The claim is proven by induction on j, from k down to 1. Note first process s_k was the only process which accessed column s_k, and thus s_k's index exiting column s_k was 1. Since no process with an input name corresponding to any columns between s_k and s_{k-1} was active, process s_k entered each of these columns from the bottom and exited with index 1.

Given the claim for s_k through s_j, we must prove the claim for the column with index s_{j-1}. By induction, the indices of s_k through s_j exiting columns preceding s_{j-1} are at most $2k-2j+1$. By part 1 of Lemma 1, their indices exiting column s_{j-1} are at most $2k-2j+1+2$ $= 2k-2(j-1)+1$. Also by part 1 of Lemma 1, process s_{j-1} computes index at most $2k-2j+2 = 2k-2(j-1)$ in column s_{j-1}. Since no process with an input name corresponding to any columns between s_{j-1} and s_{j-2} was active, each of the $k-j+1$ processes s_k through s_{j-1} entered these columns from the bottom, and retained the same index. The lemma follows. ■

Lemma 3 *Let α be a complete execution, and let snap $= \{s_m, \ldots, s_1\}$ be a snapshot returned in α, where $s_m > \ldots > s_1$. Then for all j, $m \geq j \geq 1$, the indices computed by processes s_m through s_j in column s_j are each at most $2s_m - 2s_j + 1$.*

Proof: The claim is proven by induction on j, from m down to 1. Note first that if an active process p is not in *snap*, then the snapshot taken by p in α is a superset of *snap*. It follows that every active process either has input name smaller than s_m, or observed s_m in its snapshot. Hence, every process entering column s_m, including s_m, will enter column s_m from above, and s_m's index exiting column s_m was 1.

Given the claim for s_m through s_j, we must prove the claim for the column with index s_{j-1}. By induction the indices exiting column s_j of processes s_m through s_j are at most $2s_m - 2s_j + 1$. By part 1 of Lemma 1, their indices exiting column $s_{j-1} = 2s_m - 2s_j + 1 + 2(s_j - s_{j-1})$ $= 2s_m - 2s_{j-1} + 1$.

It remains to argue that process s_{j-1} has index at most $2s_m - 2s_{j-1} + 1$ exiting column s_{j-1}. As we noted above, any process p with index not in *snap* returned a snapshot with *snap* as a subset, and hence p observed

s_{j-1}, and so entered column s_{j-1} from the top (if indeed p entered this column at all). So only s_m through s_j could have entered column s_{j-1} from the bottom, setting *down* bits possibly as high as their index upon exiting the previous column, column $s_{j-1}+1$. As in the argument above, the index of these processes exiting column $s_{j-1}+1$ was at most $2s_m - 2s_{j-1} - 1$. By part 1 of Lemma 1, process s_{j-1} has index at most $2s_m - 2s_{j-1}$ exiting column s_{j-1}. The lemma follows. ■

Lemma 4 *Let α be a complete execution, let c, p, and q be processes such that p entered column c from the top and q entered column c at the bottom. Then p entered c at the reflector network[c][$2s-2c-1$], where $2s-2c-1$ is greater than or equal to q's index exiting column $c+1$.*

Proof: Process p computed a snapshot, $snap_p = \{s_m, ..., s_1\}$ (where $s_m > ... > s_1$), which contains c. Process q computed a snapshot $snap_q = \{r_{m'}, ..., r_1\}$ (where $r_{m'} > ... > r_1$), which does not contain c. Then $snap_q \subset snap_p$, and in particular, $r_{m'} \leq s_m$. Since q entered column c, $q \geq c$, and $q \in snap_q$. It follows that $s_m > c$, and process p entered c at reflector $network[c][2s_m - 2c - 1]$. Let r_x be the smallest element of $snap_q$ which is larger than c. Note that $q \geq r_x$. Then by Lemma 3, q's index exiting r_x was at most $2r_{m'} - 2r_x + 1$. By part 1 of Lemma 1, q's index exiting column $c+1$ was at most $2r_{m'} - 2r_x + 1 + 2(r_x$-(c+1))$=2r_{m'} - 2c - 1 \leq 2s_m - 2c - 1$. The lemma follows. ■

Lemma 5 *Let α be a complete execution, let c be a column index, and let $snap = \{s_m, ..., s_1\}$ (where $s_m > ... > s_1$) be the largest snapshot returned in α that does not include c. If network[c][i].down, then $i \leq 2s_m - 2c - 1$*

Proof: Suppose $network[c][i].down$ for some i. Let s_j be the least element of $snap$ such that $s_j \geq c$, if such an element exists. Any process other than s_m through s_j either has input name less than c, or observed c in its snapshot, and entered column c from the top. Only s_m through s_j could have entered c from the bottom, setting *down* bits as high as their index exiting column $c+1$. (Since $network[c][i].down$, this means s_j exists, i.e. $s_m > c$.) By Lemma 3, their index exiting column s_j was at most $2s_m - 2s_j + 1$, and by Lemma 1, their index exiting column $c+1$ was at most $2s_m - 2s_j + 1 + 2(s_j - (c+1)) = 2s_m - 2c - 1$. ■

Lemma 6 *Let α be an execution, and let c be a column index. If network[c][i].up and network[c][j].down then $i \geq j$.*

Proof: Suppose $network[c][i].up$, $network[c][j].down$ and $i < j$. Since the *down* bits in a column are set by each processor from the bottom up, it follows that the *down* bits below j are also set at this point, particularly $network[c][i].down$ and $network[c][i+1].down$. By Lemma 4, any process entering column c from the top started above entry i, setting the *up* bits from the top down. Hence, $network[c][i].up$ also implies $network[c][i+1].up$. Moreover, there exists a process p, such that p wrote $network[c][i].down :=$ true and $network[c][i+1].down :=$ true, and there exists a process q, such that q wrote $network[c][i+1].up :=$ true and $network[c][i].up :=$ true. Now, p set $network[c][i].down$ and then read $network[c][i+1].up =$ false, so this write by p preceded q's setting of $network[c][i+1].up$. But similarly, q set $network[c][i+1].up$ before reading $network[c][i].down =$ false, so q's write to $network[c][i+1].up$ preceded p's write to $network[c][i].down$, a contradiction. The lemma follows. ■

Lemma 7 *Let α be a complete execution, and let c be a column index. If two processes compute different indices in column $c+1$ then they exit column c with distinct indices.*

Proof: If c is not the input name of any process that took steps in α, then no *up* bit is set in column c, every process retained its index from column $c+1$, and the result follows by induction. So assume that process c performed a rename operation in α.

Suppose that two processes p_j and p_{j-2} entered column c from column $c+1$ with indices j and $j-2$, and that both exited with index j from column c. (Because each process entering from an earlier column either kept its index or adds two to it, this is the only possible conflict between two such processes.) Then process p_j read $network[c][j].up =$ false, and p_{j-2} read $network[c][j-2].up =$ true when computing their indices in column c.

There are two cases, depending on whether p_j saw c in its snapshot operation. If so, then p_j entered column c from the top. Since p_j appeared in its own snapshot, p_j entered column c at or above $network[c][j]$, and since p_j later read $network[c][j].up =$ false, p_j read $network[c][j'].down =$ true for some $j' \geq j$. But this and $network[c][j-2].up =$ true contradict Lemma 6.

So assume that p_j did not see c in its snapshot operation. Then p_j entered column c from the bottom, moving from $network[c][1]$ to $network[c][j]$, finding each *up* bit false (since it later reads $network[c][j].up =$ false), and setting every *down* bit to true in each reflector through at least $network[c][j-1]$. But again this and $network[c][j-2].up =$ true contradict Lemma 6.

It follows that processes entering column c from column $c+1$ with distinct indices exit column c with distinct indices. ■

Lemma 8 *Let α be a complete execution, and let c be a column index. If the process with input name c executed* *a renaming operation, it exited column c with an index that is distinct from that of any other process exiting column c.*

Proof: There are several cases to consider. First, c may have found all *down* bits it examined to be false and computes the index 1. For this case, we consider possible conflicts with processes exiting column $c+1$ with index 1. Second, c may have found $network[c][i-1].down$ = true for some i and exited with index i. We consider possible conflicts with processes entering from column $c+1$ with index $i-2$ and i.

- Process c computed index 1 after setting $network[c][1].up$:= true and finding $network[c][1].down$ = false. Suppose another process p entered c with index 1. If p read $network[c][1].up$ = true then it computed index 3, so suppose it read $network[c][1].up$ = false. If p entered column c from the bottom, it set $network[c][1].down$:= true and read $network[c][1].up$ = false. We have c's setting $network[c][1].up$:= true before reading $network[c][1].down$ = false, which in turn precedes p's $network[c][1].down$:= true which precedes reading $network[c][1].up$ = false. But this last read by p in turn precedes c's assignment $network[c][1].up$:= true, a contradiction. If p entered column c from the top, it must not have set $network[c][1].up$:= true, but read $network[c][j].down$ = true for some $j \geq 1$. By Lemma 6, $j = 1$, and we have the same contradiction.

- Suppose then that c computes index $i > 1$, by observing $network[c][i-1].down$ = true. If $i > 2$, by Lemma 6 $network[c][i-2].up$ = false after α, so a process entering with index $i-2$ will not compute index i.

  The only other possible conflict, for $i > 1$, is with a process p entering from column $c+1$ with index i, and finding $network[c][i].up$ = false when computing its index. Let $snap_p = \{s_m, ..., s_1\}$ be the snapshot taken by process p, where $s_m > ... > s_1$.

  If p did not observe c in $snap_p$, it entered column c from the bottom, setting *down* bits until it reached reflector i or observed an *up* bit set to true. By Lemma 4, every process (including c) that entered column c from the top entered at or above reflector i. So if p observed $network[c][j].up$ = true for $j < i$, then at that point $network[c][i].up$ = true. But p later read this as false when computing its index, a contradiction. Hence, p read every *up* bit false up through reflector i, and in particular set $network[c][i].down$ to true before reading $network[c][i].up$ = false. But c set $network[c][i].up$ = true before reading $network[c][i].down$ = false, leading to the usual precedence cycle.

  It follows that p did observe c in $snap_p$. Since c observed $network[c][i-1].down$ = true, some process q such that $q > c$ left column $c+1$ with index at least $i-1$ and entered column c from the bottom, setting the *down* bits as high as reflector $i-1$. Let $snap_q = \{r_{m'}, ..., r_1\}$ be the snapshot taken by process q, where $r_{m'} > ... > r_1$. Since c is in $snap_p$ and not $snap_q$, $q \in snap_q \subset snap_p$. Let s_j be the smallest element of $snap_p$ that is bigger than c. By Lemma 3, p's index leaving column s_j is at most $2s_m - 2s_j + 1$, and by part 1 of Lemma 1, p's index, i, leaving column $c+1$ is at most $2s_m - 2s_j + 1 + 2(s_j - (c+1)) = 2s_m - 2c - 1$. But this is exactly the reflector at which p entered column c. Since p later read $network[c][i].up$ = false, it must not have set $network[c][i].up$ to true. Hence it read $network[c][j].down$ = true for some $j \geq i$, and at the time of that read, $network[c][i].down$ = true. In turn, this means some process with incoming index i or higher entered column c from the bottom, and again by Lemma 3, c entered column c at or above reflector i. Since it later read $network[c][i].down$ = false, c must have found all *down* bits down to i set to false, and in particular set $network[c][i].up$ = true.

  We have again that c set $network[c][i].up$ = true and later read $network[c][i].down$ = false, and $network[c][i].down$ was true before p read $network[c][i].up$ = false. But the later read by p preceded the assignment by c, and the read by c preceded the assignment to $network[c][i].down$, a contradiction. It follows that p must observe $network[c][i].up$ = true, and compute $i+2$ as its index leaving column c. The lemma follows.

Lemma 9 *Let α be a complete execution, and let c be a column index. Then all processes with input names c or greater which invoke renaming operations exit column c with distinct indices.*

Proof: The proof is by induction on columns, from column $f(k)$ down to 1. Since input names are unique, at most one process (the one with input name $f(k)$) exited column $f(k)$. So assume that all processes with input names $c+1$ or greater exited column $c+1$ with distinct indices. We must show this property holds also for column c. By Lemma 7, all the processes with input names greater than c exited column c with distinct indices. By Lemma 8, if the process with input name c invoked a renaming operation, it exited column c with an index that is distinct from that of any other process exiting column c. The lemma follows. ■

References

[AADGMS] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, Atomic snapshots of shared memory, *Journal of the ACM*, 40(4):873-890, 1993.

[ADT95] Afek, Y., Dauber, D. and Touitou, D. Wait-free made fast, Proceedings of the 23rd Annual ACM Symposium on the Theory of Computing, 538-547, May 1995.

[AH90] J. Aspnes and M. Herlihy, Fast randomized consensus using shared memory, *Journal of algorithms*, 441-461, September 1990.

[ABND$^+$90] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment, *Journal of the ACM* 37(3)524-548, July 1990.

[AF98] Attiya, H. and Fouren, A. Adaptive wait-free algorithms for lattice agreement and renaming, In *Proc. 17th Annual ACM Symp. on Principles of Distributed Computing*, 277-286, 1998. (Extended version available as Technion Computer Science Department Technical Report #0931, April 1998.)

[BD89] A. Bar-Noy and D. Dolev, Shared memory versus message-passing in an asynchronous distributed environment, In *Proc. 8th Annual ACM Symp. on Principles of Distributed Computing*, 307-318, 1989.

[BP89] J. Burns and L. Peterson, The ambiguity of choosing, In *Proc. 8th Annual ACM Symp. on Principles of Distributed Computing*, 41-51, 1989.

[G98] E. Gafni, Public communication, 17th Annual ACM Symp. on Principles of Distributed Computing, July 1998, Puerto Vallarto, Mexico, July 1998.

[G92] E. Gafni, More about renaming: Fast algorithm and reduction to the k-set Test-and-Set problem. Unpublished manuscript, 1992.

[GLS95] R. Gawlick, N. Lynch, and N. Shavit, Concurrent timestamping made simple, In *Proc. of the 1st Israel Symp. on the Theory of Computing and Systems*, Springer-Verlag, 171-185, May 1992.

[HS93] M. Herlihy and N. Shavit. The asynchronous computability theorem for t-resilient tasks. In *Proc. 25th ACM Symp. on Theory of Computing*, 111–120, May 1993.

[Lam87] L. Lamport. A fast mutual exclusion algorithms. *ACM Trans. on Computer Systems*, 5(1):1–11, February 1987.

[MA95] M. Moir and J. Anderson, Wait-free algorithms for fast, long-lived renaming, *Science of Computer Programming* 25(1):1-39, October 1995.

[M] M. Moir, Fast, long-lived renaming improved and simplified, *Science of Computer Programming*, 30(3):287-308, March 1998.