ADAPTIVE AND EFFICIENT ALGORITHMS FOR LATTICE AGREEMENT AND RENAMING*

HAGIT ATTIYA† AND ARIE FOUREN†

Abstract. In a shared-memory system, n independent asynchronous processes, with distinct names in the range $\{0, \ldots, N-1\}$, communicate by reading and writing to shared registers. An algorithm is *wait-free* if a process completes its execution regardless of the behavior of other processes. This paper considers wait-free algorithms whose complexity adjusts to the level of contention in the system: An algorithm is *adaptive* (to total contention) if its step complexity depends only on the actual number of active processes, k; this number is unknown in advance and may change in different executions of the algorithm.

Adaptive algorithms are presented for two important decision problems, *lattice agreement* and $(6k-1)$-*renaming*; the step complexity of both algorithms is $O(k \log k)$. An interesting component of the $(6k-1)$-renaming algorithm is an $O(N)$ algorithm for $(2k-1)$-renaming; this improves on the best previously known $(2k-1)$-renaming algorithm, which has $O(Nnk)$ step complexity.

The efficient renaming algorithm can be modified into an $O(N)$ implementation of atomic snapshots using *dynamic single*-writer multi-reader registers. The best known implementations of atomic snapshots have step complexity $O(N \log N)$ using *static single*-writer multi-reader registers, and $O(N)$ using *multi*-writer multi-reader registers.

Key words. shared-memory systems, wait-free computation, atomic read/write registers, renaming, lattice agreement, atomic snapshots

AMS subject classifications. 68P05, 68Q10, 68Q20, 68Q22

1. Introduction. An asynchronous shared-memory system contains n processes running at arbitrary speeds and communicating by reading from and writing to shared registers; processes have distinct names in the range $\{0, \ldots, N-1\}$, $n \leq N$. In a *wait-free* algorithm, a process terminates in a finite number of steps, even if other processes are very slow, or even stop taking steps completely.

The step complexity of many wait-algorithms depends on N; for example, collecting up-to-date information from all processes typically requires to read an array indexed with processes' names. Real distributed systems need to accommodate a large number of processes, i.e., N is large, while often only a small number of processes take part in the computation. For such systems, step complexity depending on n or N is undesirable; it is preferable to have step complexity which adjusts to the number of processes participating in the algorithm.

An algorithm is *adaptive* (to total contention) if its step complexity depends only on the total number of processes participating in the algorithm, denoted k; k is unknown in advance and it may change in different executions of the algorithm. The step complexity of an adaptive algorithm adjusts to the number of active processes: It is constant if a single process participates in the algorithm, and it gradually grows as the number of active processes increases.

A weaker guarantee is provided by *range-independent* algorithms whose step complexity depends only on n, the maximal number of processes; clearly, n is fixed for all executions.[1] The advantage of range-independent algorithms is quite restricted: They require *a priori* knowledge of n, which is often difficult to determine; moreover, their

[1]An extended abstract of this paper appeared in proceedings of the *17th ACM Symposium on Principles of Distributed Computing* (June 1998), pp. 277–286.

[2]Department of Computer Science, The Technion, Haifa 32000, Israel (hagit@cs.technion.ac.il, leonf@cs.technion.ac.il). Supported by the fund for the promotion of research at the Technion.

[1]Moir and Anderson [27] use the term "fast", which conflicts with other papers [3, 25].

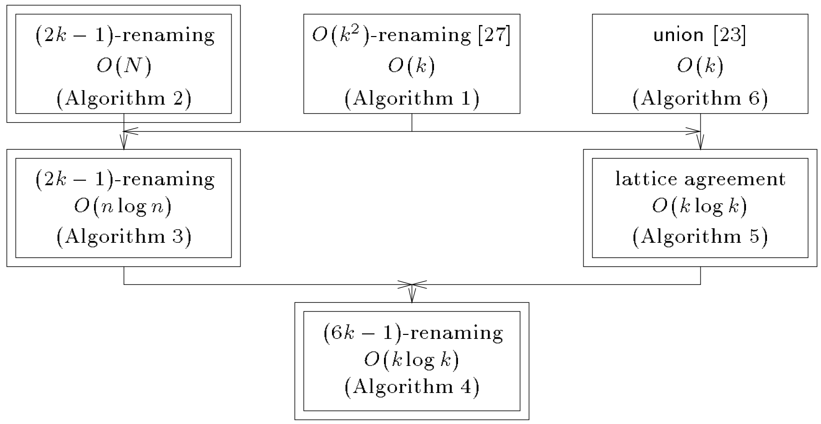

FIG. 1. *The algorithms presented in this paper; double boxes indicate the main results.*

step complexity is not optimal when the actual number of participating processes is much lower than the upper bound. Yet, as we show, they can be useful tools in the construction of adaptive algorithms.

This paper presents adaptive wait-free algorithms for lattice agreement and renaming, using only read and write operations. Along the way, we improve the step complexity of non-adaptive algorithms for renaming. Figure 1 depicts the algorithms presented in this paper.

In the one-shot M*-renaming* problem [10], processes are required to choose distinct names in a range of size $M(k)$, for some bounded function M. This paper does not consider the more general *long-lived* renaming problem [9], in which processes repeatedly *acquire* and *release* names. Adaptive renaming can serve as an intermediate step in adaptive algorithms for other problems [9, 26, 27, 28]: The new names replace processes' original names, making the step complexity depend only on the number of active processes. Our algorithms employ this technique, as well as [6, 7].

An efficient adaptive algorithm for renaming could not be derived from known algorithms: The best previously known algorithm for renaming with linear name space [18] has $O(Nnk)$ step complexity, yielding $O(k^3)$ step complexity (at best) if it can be made adaptive. Thus, we first present an $(2k-1)$-renaming algorithm with $O(N)$ step complexity, which is neither adaptive nor range-independent. This algorithm is based on a new "shrinking network" construction, which we consider to be the novel algorithmic contribution of our paper.

The new linear renaming algorithm is employed in a range-independent algorithm for $(2k-1)$-renaming with $O(n \log n)$ step complexity. Processes start with an adaptive $O(k^2)$-renaming algorithm whose step complexity is $O(k)$; this is a simple modification of the range-independent renaming algorithm of Moir and Anderson [27]. Then, processes reduce the range of names in $O(\log n)$ iterations; each iteration uses our new linear renaming algorithm.

The range-independent renaming algorithm is used to construct an adaptive $(6k-1)$-renaming algorithm with $O(k \log k)$ step complexity. In this algorithm, processes are partitioned into $O(\log k)$ disjoint sets according to views obtained from an adaptive lattice agreement algorithm (described below). This partition bounds the number of processes in each set, and allows them to employ a range-independent $(2k-1)$-

renaming algorithm designed for this bound. Different sets use disjoint name spaces; no coordination between the sets is required.

In the *lattice agreement* problem [15], processes obtain comparable (by containment) subsets of the set of active processes. A wait-free lattice agreement algorithm can be turned into a wait-free implementation of an *atomic snapshot* object, with $O(n)$ additional read/write operations [15]. Atomic snapshot objects allow processes to get instantaneous global views ("snapshots") of the shared memory and thus, they simplify the design of wait-free algorithms.

The step complexity of our adaptive algorithm for lattice agreement is $O(k \log k)$. In this algorithm, processes first obtain names in a range of size $O(k^2)$ using the simple algorithm with $O(k)$ step complexity. Based on its reduced name, a process enters an adaptive variant of the tree used in the lattice agreement algorithm of Inoue et al. [23].

Appendix C describes how the shrinking network is modified to get a lattice agreement algorithm with $O(N)$ step complexity, using *dynamic single*-writer single-reader registers; this gives an implementation of atomic snapshots with the same complexity. Previous implementations of atomic snapshots had either $O(N \log N)$ step complexity using *static single*-writer multi-reader registers [16], or $O(N)$ step complexity using *multi*-writer multi-reader registers [23].

The renaming problem was introduced and solved by Attiya et al. [10] for the message-passing model; Bar-Noy and Dolev [17] solved the problem in the shared-memory model. Burns and Peterson [19] considered the *l-assignment* problem—dynamic allocation of l distinct resources to processes. They present a wait-free l-assignment algorithm which assumes $l \geq 2k-1$, where k is the number of processes trying to acquire a resource. All these algorithms have exponential step complexity [21]. Borowsky and Gafni [18] present an algorithm for one-shot $(2k-1)$-renaming using $O(Nnk)$ read/write operations.

Anderson and Moir [9] define long-lived renaming and present range-independent algorithms for one-shot and long-lived renaming; their algorithms use test&set operations. Moir and Anderson [27] introduced a building block, later called a *splitter*, and employ it in range-independent algorithms for long-lived renaming, using read/write operations. Moir and Garay [28, 26] give a range-independent long-lived $O(kn)$-renaming algorithm, using only read/write operations. By combining with a long-lived $(2k-1)$-renaming algorithm [19] they obtain a range-independent long-lived $(2k-1)$-renaming algorithm; its step complexity is dominated by the exponential step complexity of Burns and Peterson's algorithm.

Herlihy and Shavit [22] show that one-shot renaming requires $2k-1$ names. This implies that our range-independent renaming algorithm provides an optimal name space. The name space provided by our adaptive renaming algorithm is not optimal ($M = 6k-1$); still, it is linear in the number of active processes.

Following the original publication of our paper [12], Afek and Merritt [4] used our algorithms to obtain an adaptive wait-free $(2k-1)$-renaming algorithm, with $O(k^2)$ step complexity.

In another paper [13], we present an adaptive collect algorithm with $O(k)$ step complexity and derive adaptive algorithms for atomic snapshots, immediate snapshots and $(2k-1)$-renaming. That paper emphasizes the modular use of a collect operation to make known algorithms adaptive; the algorithms have higher step complexity than those presented here.

Our algorithms adapt to the total number of participating processes, that is, if a process ever performs a step then it influences the step complexity of the algorithm throughout the execution. More useful are algorithms which adapt to the *current* contention and whose step complexity decreases when processes stop participating. Afek, Dauber and Touitou [3] present implementations of long-lived objects which adapt to the current contention; they use load-linked and store-conditional operations. Recent papers present algorithms for long-lived renaming [2, 14], collect [6] and snapshots [7] which adapt to the current contention using only read/write operations.

Lamport [25] suggests a *mutual exclusion* algorithm which requires a constant number of steps when a single process wishes to enter the critical section, using read/write operations; when several processes compete for the critical section, the step complexity depends on the range of names. Alur and Taubenfeld [8] show that this behavior is inherent for mutual exclusion algorithms. Choy and Singh [20] present mutual exclusion algorithms whose *time complexity*—the time between consecutive entries to the critical section—is $O(k)$, using only read/write operations. Afek, Stupp and Touitou [6] use an adaptive collect algorithm to derive an adaptive version of the Bakery algorithm [24]; they present another mutual execlusion algorithm in [5]. Recently, Attiya and Bortnikov [11] presented a mutual exclusion algorithm whose time complexity is $O(\log k)$; this algorithm employs an unbalanced tournament tree with the same structure as our adaptive lattice agreement tree.

2. Preliminaries.

2.1. The Model. In the shared-memory model, *processes* $p_0, \ldots, p_{n-1}$ communicate by applying *operations* on shared objects. A process p_i is modeled as a (possibly infinite) state machine; process p_i has a distinct name $id_i \in \{0, \ldots, N-1\}$, $n \leq N$.

The shared objects considered in this paper are *atomic read/write registers*, accessed by read and write operations. A read(R) operation does not change the state of R, and returns the current value stored in R; a write(v,R) operation changes the state of R to v. A *multi-writer multi-reader* register allows any process to perform read and write operations. A *single-writer multi-reader* register allows only a single process to perform write operations, and any process to perform read operation. A single-writer multi-reader register is *dynamic* if the identity of the single process writing to the register varies in different executions; otherwise, it is *static*.

An *event* is a computation step by a single process; the process determines the operation to perform according to its state, and its next state according to its state and the value returned by the operation. Computations in the system are captured as sequences of events. An *execution* α is a (finite or infinite) sequence of events $\phi_0, \phi_1, \phi_2, \ldots$. For every $r = 0, 1, \ldots$, if p_i is the process performing the event ϕ_r, then it applies a read or a write operation to a single register and changes its state according to its transition function. There are no constraints on the interleaving of events by different processes, reflecting the assumption that processes are *asynchronous* and there is no bound on their relative speeds.

Consider an execution α of some algorithm A. For process p_i, $step(A, \alpha, p_i)$ is the number of read/write operations p_i performs in α. The step complexity of A in α, denoted $step(A, \alpha)$, is the maximum of $step(A, \alpha, p_i)$ over all processes p_i. A process is *active* in α if it takes a step in α, that is, $step(A, \alpha, p_i) \neq 0$; $k(\alpha)$ denotes the number of active processes in α.

Algorithm A is *range-independent* if there is a function $f : \mathcal{N} \mapsto \mathcal{N}$ such that in every execution α of A, $step(A, \alpha) \leq f(n)$. Namely, the step complexity of A in every execution is bounded by a function of the total number of processes (which is known in advance); it does not depend on the range of the initial names.

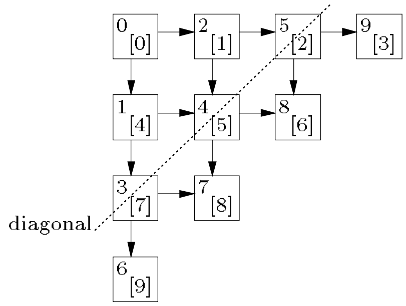

FIG. 2. *The grid used for $O(k^2)$-renaming (depicted for $n = 4$).*

Algorithm A is *adaptive (to total contention)* if there is a function $f : \mathcal{N} \mapsto \mathcal{N}$ such that in every execution α of A, $step(A, \alpha) \leq f(k(\alpha))$. Namely, the step complexity of A in α is bounded by a function of the number of active processes in α. Clearly, the number of active processes is not known *a priori*.

A *wait-free* algorithm guarantees that every process completes its computation in a finite number of steps, regardless of the behavior of other processes. Since $k(\alpha)$ is bounded (by n) it follows that adaptive algorithms are wait-free.

2.2. Problems. The *M-renaming* problem [10] requires processes to choose distinct names in a range that depends only on the number of active processes. Namely, there is a function $M : \mathcal{N} \mapsto \mathcal{N}$ such that in every execution α, processes output distinct names in the range $\{0, \ldots, M(k(\alpha)) - 1\}$.

In the *lattice agreement* problem [15], every process p_i outputs V_i a subset of the active processes (e.q., a *view*) such that the following conditions hold:

Self-inclusion: $p_i \in V_i$, for every i.

Comparability: V_i and V_j are comparable (either $V_i \subseteq V_j$ or $V_j \subseteq V_i$), for every i and j.

2.3. Simple $O(k^2)$-Renaming in $O(k)$ Operations. The first step in our algorithms is a simple adaptive $O(k^2)$-renaming algorithm. This algorithm reduces the range of names to depend only on the number of active processes; later stages use these distinct names, without sacrificing the adaptiveness. We describe this algorithm first since it employed in both adaptive algorithms presented in this paper.

The basic building block of this algorithm is the *splitter* of Moir and Anderson [27]. A process executing a splitter obtains **down**, **right** or **stop**. At most one process obtains **stop** and when a single process executes the splitter it obtains **stop**; when two or more processes execute the splitter, not all of them obtain the same value. In this way, the set of processes accessing the splitter is "split" into smaller subsets.

As in [27], splitters are arranged in a grid of size $n \times n$ (Figure 2). A process starts at the upper left corner of the grid; the splitters direct the process either to continue (moving right or down in the grid), or to obtain the number associated with the current splitter. The grid spreads the processes so that each process eventually stops in a distinct splitter.

The difference between the algorithm of Moir and Anderson [27] and our algorithm is that they number splitters by by *rows*, while we number splitters by *diagonals*. Splitter (i, j), in row i and column j, $0 \leq i < n$ and $0 \leq j < n$, is numbered $(i + j)(i + j + 1)/2 + j$. Figure 2 shows our numbering; the numbering of Moir and Anderson appears in square brackets.

Algorithm 1 Adaptive $k(k+1)/2$-renaming.

```
Procedure Adaptive_k(k+1)/2-renaming()
private i, j: integer, initially 0                          // row and column indices
private move: {down, right, stop}, initially down           // direction
while ( move ≠ stop ) do
    move := Splitter[i, j]()          // execute splitter in grid position (i, j)
    if ( move = down ) then i++                              // increase row
    if ( move = right ) then j++                             // increase column
return((i + j)(i + j + 1)/2 + j)   // get a name based on the current splitter

Procedure Splitter[i, j]                          // from Moir and Anderson [27]
    shared X[i, j]: {⊥} ∪ {0, ..., N − 1}, initially ⊥
    shared Y[i, j]: Boolean, initially false
X[i, j] := id
if ( Y[i, j] ) then return(right)
else Y[i, j] := true
    if ( X[i, j] = id ) then return(stop)
    else return(down)
```

Algorithm 1 presents pseudocode for the grid and for a splitter.[2]

We say that splitter (i, j) is $i + j$ *steps away* from splitter $(0, 0)$, the top left corner of the grid. As shown in [27, Section 3.1], if k processes access the grid then each process stops after $O(k)$ operations in a distinct splitter which is at most $k - 1$ steps away from $(0, 0)$. A simple counting argument shows that these splitters have numbers in the range $\{0, \ldots, k(k+1)/2 - 1\}$.

THEOREM 2.1. *Algorithm 1 solves $k(k+1)/2$-renaming with $O(k)$ step complexity.*

3. $(2k-1)$-Renaming in $O(N)$ Operations. As explained in the introduction, the step complexity of adaptive renaming depends on a new linear renaming algorithm, which is neither range-independent nor adaptive.

The algorithm is organized as a network of *reflectors*. A reflector has two distinguished entrances; a process accessing the reflector changes the direction of its movement if another process accessed the reflector, depending on the entrance through which it entered the reflector.

The network consists of N columns, numbered $0, \ldots, N-1$, from left to right (see Figure 3). Column $c = 0, \ldots, N-1$ contains $2c - 1$ reflectors, numbered $c, c-1, \ldots, 0, -(c-1), -c$, from top to bottom. Process q with name c starts at the topmost reflector of column c and descends through column c, until it sees another process accessing the same reflector. Then, q moves left to right towards column $N-1$; q outputs the row on which it exits column $N-1$.

For column c, S_{c-1} is the set of processes starting in columns $0, \ldots, c-1$. The main property of the network is that processes in S_{c-1} enter column c on distinct rows among the lowest $2|S_{c-1}| - 1$ ones. Therefore, processes in S_{c-1} do not access the same reflectors in column c (or larger); they may interfere only with the single process descending through column c.

[2] Algorithms declare private variables only if their usage in not obvious, or their initial value is important.

Process q descends through column c until it accesses a reflector in row r through which a process in S_{c-1} has passed; then, q moves to column $c+1$, remaining in row r. If process $p \in S_{c-1}$ accesses a reflector which q has passed, then p moves one row up to column $c+1$; if p accesses a reflector which q did not pass, then p moves one row down to column $c+1$. Therefore, processes in S_{c-1} which enter column c on rows $> r$, move one row up; processes in S_{c-1} which enter column c on rows $< r$, move one row down. Process q leaves on one of the free rows between the rows occupied by these two subsets of S_{c-1} (Figure 4(c)). Thus, processes in $S_c = S_{c-1} \bigcup \{q\}$ leave column c on distinct rows. Since the new names of the processes are the rows on which they leave the network, they output distinct names.

The interaction of processes in column c guarantees that processes in S_{c-1} move to upper rows in column $c+1$ only if q is active; at most two additional rows are occupied (Figure 5(b)). If q is not active, then processes leave column c exactly on the same number of rows as they enter (Figure 5(a)). Thus, an active process causes at most two rows to be occupied; if there are k active processes, then they leave the network on the lowest $2k-1$ rows.

More formally, a reflector has two *entrances*, in_0 and in_1, two *lower exits*, $down_0$ and $down_1$, and two *upper exits*, up_0 and up_1. A process entering the reflector on entrance in_i leaves the reflector only on exits up_i or $down_i$ (see top left corner of Figure 3). If a single process enters the reflector then it must leave on a lower exit, and at most one process leaves on a lower exit; it is possible that two processes entering the reflector will leave on upper exits. A reflector is easily implemented with two Boolean registers (see Algorithm 2).

The reflectors of column c, denoted $S[c,-c], \ldots, S[c,0], \ldots, S[c,c]$, are connected as follows:

- The upper exit up_0 of $S[c,r]$ is connected to entrance in_0 of $S[c+1,r+1]$.
- The upper exit up_1 of $S[c,r]$ is connected to entrance in_0 of $S[c+1,r]$.
- The lower exit $down_0$ of $S[c,r]$ is connected to entrance in_0 of $S[c+1,r-1]$.
- The lower exit $down_1$ of a reflector $S[c,r]$ is connected to entrance in_1 of reflector $S[c,r-1]$, if $r > -c$; if $r = -c$ (the lowest reflector of column c), then it is connected to entrance in_0 of reflector $S[c+1,r-1]$.

In Algorithm 2, a process with name c starts on entrance in_1 of the upper reflector of column c; it descends through column c (leaving on exit $down_1$) until it sees another process or it reaches the bottom of the column. At this point, the process leaves on exit up_1 to the next column, and moves towards column $N-1$; in each column y, it enters exactly one reflector on entrance in_0; it leaves on exit up_0 if it sees another process, or on exit $down_0$, otherwise.

Suppose that p_j enters the reflector on entrance in_i, $i \in \{0,1\}$, and no process enters the reflector on entrance in_{1-i}. Since no process writes to R_{1-i}, p_j reads **false** from R_{1-i} and leaves the reflector on the lower exit, $down_i$. This implies the following lemma:

LEMMA 3.1. *If a single process enters a reflector, then it leaves on a lower exit.*

Similar arguments are used in the proof of the next lemma:

LEMMA 3.2. *If a single process enters a reflector on in_0 and a single process enters the reflector on in_1, then at most one process leaves the reflector on a lower exit.*

Proof. Assume that p_i enters the reflector on in_0 and p_j enters the reflector on in_1. If both processes read **true** from R_1 and R_0, then by the algorithm, $exit(p_i) = up_0$, $exit(p_j) = up_1$, and the lemma holds. Otherwise, without loss of generality, p_i reads

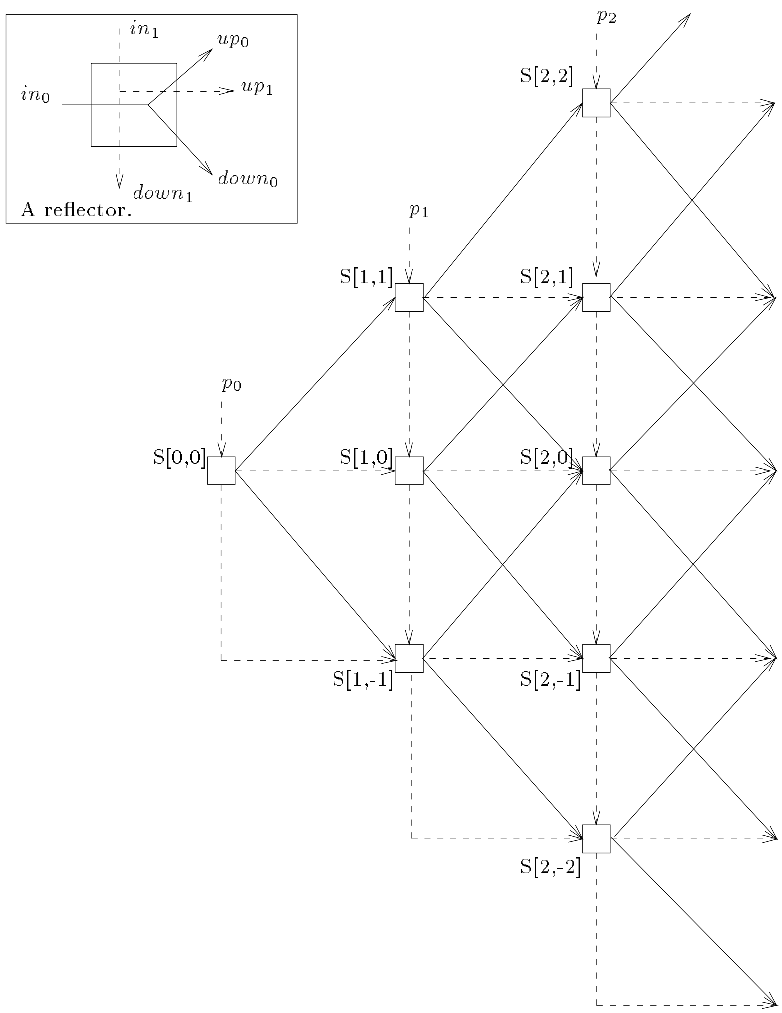

FIG. 3. *The network of reflectors for* $(2k-1)$*-renaming (depicted for* $n = 3$*).*

false from R_1. Since p_i reads **false** from R_1, p_j writes to R_1 in Line 1 after p_i reads R_1 at Line 2. Therefore, p_j reads R_0 in Line 2 after p_i writes to R_0 in Line 1. Consequently, p_j obtains **true** from R_0 and by the algorithm, $exit(p_j) = up_1$, which proves the lemma. ☐

Recall that S_c contains the active processes starting on columns $0, \ldots, c$. For every process $p_i \in S_c$, let $row(p_i, c)$ be the value of the local variable *row* before p_i accesses the first reflector in column $c+1$. The next lemma shows that processes exit a column on distinct rows.

LEMMA 3.3. *For every pair of processes* $p_i, p_j \in S_c$*,* $0 \leq c < N$*,* $row(p_i, c) \neq row(p_j, c)$*.*

Proof. The proof is by induction on the column c. In the base case, $c = 0$, the lemma trivially holds since only one process may access a reflector in column 0.

For the induction step, suppose that the lemma holds for column $c \geq 0$; there are two cases:

Algorithm 2 $(2k-1)$-renaming.

```
Procedure shrink(name : 0, ..., N − 1)                 // renaming algorithm
private col, row: integer, initially name   // start on top reflector of column name
while ( col = name ) do                            // descend through column name
    exit := reflector[row,col](1)                                  // enter on in_1
    if ( exit = up_1 ) then col++
    else row−−                                                  // exit = down_1
        if ( row < −col ) then col++   // reached the lowest reflector in column
while ( col < N ) do                                // move towards column N − 1
    exit := reflector[row, col](0)                                 // enter on in_0
    if ( exit = up_0 ) then col++; row++;
    else col++; row−−;                                          // exit = down_0
return(row + N);

Procedure reflector(entrance r : 0,1)
R_r := true ;
if ( R_{1−r} = false ) then return(down_r)
else return(up_r)
```

Case 1: If no process starts on column $c+1$, then by the algorithm, no reflector in column $c+1$ is accessed on in_1. By Lemma 3.1, every process $p_i \in S_c$ leaves column $c+1$ on exit $down_0$. By the algorithm, we have

$$row(p_i, c+1) = row(p_i, c) - 1$$

(Figure 4(a)) and the lemma holds by the induction hypothesis.

Case 2: Suppose that process q starts on column $c+1$. Let $S[c+1, r']$ be the last reflector accessed by q in column $c+1$. That is, q leaves reflectors $S[c+1, c+1], \ldots, S[c+1, r'+1]$ on exits $down_1$; q does not access any of the reflectors $S[c+1, r'-1], \ldots, S[c+1, -(c+1)]$.

By Lemma 3.2, every process $p_i \in S_c$ which enters column $c+1$ on row $r'+1$ or higher, exits column $c+1$ on up_0, and we have:

$$row(p_i, c+1) = row(p_i, c) + 1 > r' + 1.$$

By Lemma 3.1, every process $p_i \in S_c$ which enters column $c+1$ on row $r'-1$ or lower, exits column $c+1$ on $down_0$, and we have:

$$row(p_i, c+1) = row(p_i, c) - 1 < r' - 1.$$

Now consider process q. By the algorithm, q leaves column $c+1$ either on exit $down_1$ of the lowest reflector in the column, $S[c+1, -(c+1)]$, or on exit up_1 of a reflector $S[c+1, r']$, where $-(c+1) \leq r' \leq c+1$.

If q leaves reflector $S[c+1, -(c+1)]$ on $down_1$ (Figure 4(b)), then by the algorithm,

$$row(q, c+1) = -(c+1) - 1 = r' - 1.$$

If there is a process $p_j \in S_c$ which also accesses $S[c+1, -(c+1)]$, then by Lemma 3.2, p_j leaves $S[c+1, -(c+1)]$ on exit up_0 and therefore,

$$row(p_j, c+1) = -(c+1) + 1 = r' + 1.$$

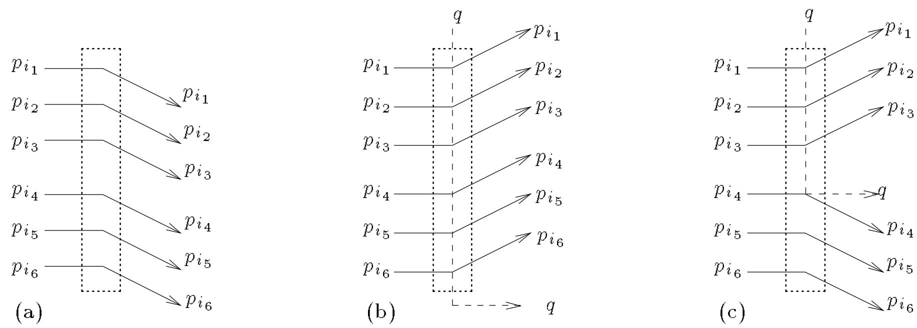

FIG. 4. *Illustration for the proof of Lemma 3.3—column* $c+1$.

If q leaves reflector $S[c+1, r']$ on up_1 (Figure 4(c)), then by the algorithm, $row(q, c+1) = r'$. By Lemma 3.2, there is a process $p_j \in S_c$ which accesses $S[c+1, r']$; that is, $row(p_j, c) = r'$. By the algorithm

$$row(p_j, c+1) = \begin{cases} r' - 1 & \text{if } p_j \text{ leaves on } down_0 \\ r' + 1 & \text{otherwise} \end{cases}$$

The induction hypothesis and the above equations imply that in all cases, $row(p_i, c+1) \neq row(p_j, c+1)$, for every pair of processes $p_i, p_j \in S_c$. ∎

Therefore, processes exit the network on different rows and hence, obtain distinct names. The next lemma shows that processes in S_c leave column c on the lowest $2|S_c| - 1$ rows.

LEMMA 3.4. *For every process* $p_i \in S_c$*,* $0 \leq c < N$*,* $-(c+1) \leq row(p_i, c) < -(c+1) + 2|S_c| - 1$*.*

Proof. The proof is by induction on c. In the base case, $c = 0$. If there is a process p_i such that $id_i = 0$, then by Lemma 3.1, $exit(p_i, 0) = down_1$, since no process accesses reflector $S[0, 0]$ on in_0. Therefore, by the algorithm, we have $row(p_i, 0) = -1$, and the lemma holds.

For the induction step, suppose that the lemma holds for column $c \geq 0$; there are two cases:

Case 1: If no process starts on column $c+1$, then no process accesses reflectors in column $c+1$ on entrance in_1 (Figure 5(a)). Therefore, by Lemma 3.1, each process $p_i \in S_c$ exits column $c+1$ on $down_0$, and therefore, $row(p_i, c+1) = row(p_i, c) - 1$. By the induction hypothesis

$$row(p_i, c+1) = row(p_i, c) - 1 \geq -(c+1) - 1 = -((c+1) + 1) \ .$$

Also, since $|S_{c+1}| = |S_c|$,

$$row(p_i, c+1) = row(p_i, c) - 1 < -(c+1) + 2|S_c| - 1 - 1 = -((c+1)+1) + 2|S_{c+1}| - 1 \ .$$

The lemma follows from these inequalities.

Case 2: Suppose that process q starts on column $c+1$. By the induction hypothesis, processes from S_c access only the lowest $2|S_c| - 1$ reflectors in column $c+1$. Since no process accesses the upper reflectors $S[c+1, c+1], \ldots, S[c+1, -(c+1) + 2|S_c| - 1]$ on in_0, by Lemma 3.1, q accesses these reflectors until it reaches a reflector $S[c+1, r']$

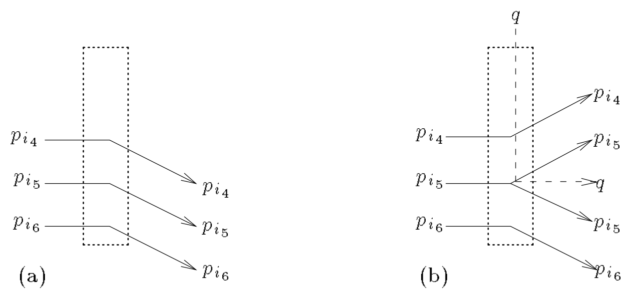

FIG. 5. *Illustration for the proof of Lemma 3.4—column $c+1$.*

accessed by another process, or until it reaches the lowest reflector $S[c+1, -(c+1)]$ in the column (Figure 5(b)). Therefore, q leaves column $c+1$ either on exit $down_1$ of reflector $S[c+1, -(c+1)]$ or on exit up_1 of a reflector $S[c+1, r']$, where $-(c+1) \le r' < -(c+1) + 2|S_c| - 1$. By the algorithm, this implies

$$-((c+1)+1) \le row(q, c+1) < -(c+1) + 2|S_c| - 1$$

Since $S_{c+1} = S_c \cup \{q\}$, $|S_{c+1}| = |S_c| + 1$. Therefore,

$$(1) \qquad -((c+1)+1) \le row(q, c+1) < -((c+1)+1) + 2|S_{c+1}| - 2$$

According to the algorithm, for each process $p_i \in S_c$,

$$row(p_i, c+1) = \begin{cases} row(p_i, c) + 1 & \text{if } exit(p_i, c+1) = up_0 \\ row(p_i, c) - 1 & \text{otherwise} \end{cases}$$

Together with the induction hypothesis, this implies

$$(2) \qquad row(p_i, c+1) \ge row(p_i, c) - 1 \ge -(c+1) - 1 = -((c+1)+1)$$

Also,

$$\begin{aligned} row(p_i, c+1) &\le row(p_i, c) + 1 \\ &< -(c+1) + 2|S_c| - 1 + 1 \\ &= -((c+1)+1) + 1 + 2(|S_{c+1}| - 1) - 1 + 1 \\ (3) \qquad &= -((c+1)+1) + 2|S_{c+1}| - 1 \end{aligned}$$

The lemma follows from inequalities (1), (2) and (3). ◻

Lemma 3.4 implies that processes leave the network on rows $-N, \ldots, -N + 2|S_{N-1}| - 2$. Since $|S_{N-1}| \le k$, the names chosen in Line 10 are in the range $\{0, \ldots, 2k-2\}$.

Process p_j accesses at most $2id_j + 1$ reflectors in column id_j, and exactly one reflector in each column $id_j + 1, \ldots, N-1$. Each reflector requires a constant number of operations, implying the next theorem:

THEOREM 3.5. *Algorithm 2 solves $(2k-1)$-renaming with step complexity $O(N)$.*

The network consists of $O(N^2)$ reflectors; each reflector is implemented with two registers. Register R_i of a reflector is written only by a process entering the reflector on entrance in_i. Entrance in_1 of a reflector is accessed only by the single process starting on this column, and entrance in_0 is accessed by at most one process (by Lemma 3.3). Therefore, we use $O(N^2)$ *dynamic* single-writer single-reader registers.

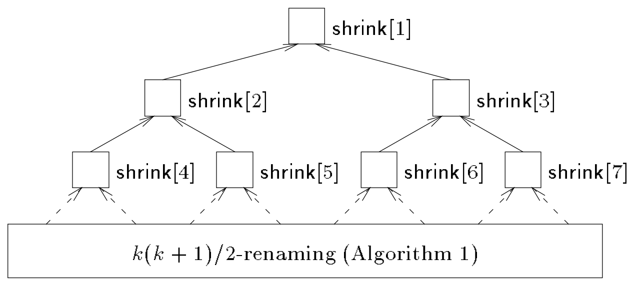

FIG. 6. *The range-independent algorithm for $(2k-1)$-renaming (depicted for $n = 3$).*

Algorithm 3 Range-independent $(2k-1)$-renaming for n processes.

```
Procedure indRenaming_n()
    temp-name := Adaptive_k(k+1)/2-renaming()                    // Algorithm 1
    ℓ := 2^h + ⌊temp-name/2⌋   // h = ⌈log n(n+1)/2⌉ − 1 is the height of the tree
    side := temp-name mod 2
    temp-name := 0
    while ( ℓ ≥ 1 ) do
        temp-name := shrink[ℓ](temp-name + side · (2n − 1))
        side := ℓ mod 2
        ℓ := ⌊ℓ/2⌋
    return(temp-name)
```

4. $(2k-1)$-Renaming in $O(n \log n)$ Operations. A range-independent $(2k-1)$-renaming can be obtained by combining adaptive $O(k^2)$-renaming and non-adaptive $(2k-1)$-renaming. First, the names are reduced into a range of size $O(n^2)$ (Algorithm 1); these names are used to enter the shrinking network of Algorithm 2, which reduces them into a range of size $(2k-1)$. The shrinking network is started with names of size $O(n^2)$, and hence, the step complexity of this simple algorithm is $O(n^2)$. The algorithm presented in this section obtains $O(n \log n)$ step complexity by reducing the name space gradually in $O(\log n)$ iterations. To do so, distinct copies of shrink (Algorithm 2) are associated with the vertices of a complete binary tree of height $\lceil \log n(n+1)/2 \rceil - 1 \approx 2 \log n$ (Figure 4). Each copy of shrink is designed for names in a range of size $4n-2$; that is, it employs a network with $4n-2$ columns.

A process starts Algorithm 3 by acquiring a name using $O(k^2)$-renaming; this name determines from which leaf to start. The process performs the shrinking network associated with each vertex v on the path from the leaf to the root, starting at a column which is determined by the name obtained at the previous vertex: If it ascends from the left subtree of v, then it starts at one of the first $2n-1$ columns of the network; otherwise, it starts at one of the last $2n-1$ columns. The process outputs the name obtained at the root.

The vertices of the tree are numbered in BFS order (Figure 4): The root is numbered 1; if vertex v is numbered ℓ, then its left child is numbered 2ℓ, and its right child is numbered $2\ell+1$. The copy of Algorithm 2 associated with a vertex numbered ℓ is denoted shrink[ℓ].

LEMMA 4.1. *For every vertex v, processes executing* shrink[v] *obtain distinct temporary names in the range $\{0, \ldots, 2k-2\}$.*

Proof. The proof is by induction on d, the height of v. In the base case, $d = 0$.

Algorithm 4 Adaptive $(6k-1)$-renaming.

$V :=$ AdaptiveLA() // Algorithm 5, presented below
$r := \lceil \log |V| \rceil$
temp-name $:=$ indRenaming$_{2^r}$() // Algorithm 3
if ($r = 0$) then return(*temp-name*)
else return(*temp-name* $+ 2^{r+1}$)

After executing Algorithm 1, processes get distinct names in the range $\{0, \ldots, k(k+1)/2-1\}$. Therefore, at most one process accesses v from the left executing shrink$[v]$ with temporary name 0, and at most one process accesses v from the right, executing shrink$[v]$ with temporary name $2n-1+0 = 2n-1$. Thus, they execute shrink$[v]$ with different temporary names in the range $\{0, \ldots, 4n-3\}$. Theorem 3.5 implies they obtain distinct names in the range $\{0, 1, 2\}$ and therefore, the lemma holds when $d = 0$.

For the induction step, assume the lemma holds for vertices at height d, and let v be a vertex at height $d+1$. By the induction hypothesis and by the algorithm, processes accessing v from the left child have distinct temporary names in the range $\{0, \ldots, 2k-2\}$, and processes accessing v from the right child have distinct names in the range $\{2n-1, \ldots, 2n+2k-3\}$. Thus, processes execute shrink$[v]$ with distinct names in the range $\{0, \ldots, 4n-3\}$, and obtain distinct names in the range $\{0, \ldots, 2k-2\}$, by Theorem 3.5. ◻

Therefore, processes obtain distinct names in the range $\{0, \ldots, 2k-2\}$ after completing shrink at the root. A process performs shrink in $h = O(\log n)$ vertices of the tree, and each vertex requires $O(n)$ operations (Theorem 3.5). This implies the following theorem:

THEOREM 4.2. *Algorithm 3 solves $(2k-1)$-renaming with $O(n \log n)$ step complexity.*

5. $(6k-1)$-Renaming in $O(k \log k)$ Operations. In our adaptive $(6k-1)$-renaming algorithm, a process estimates the number of active processes and performs a copy of the range-independent $(2k-1)$-renaming algorithm (Algorithm 3) designed for this number. Processes may have different estimates of k, the number of active processes, and perform different copies of Algorithm 3. Instead of consolidating the names obtained in the different copies, disjoint name spaces are allocated to the copies.

The number of active processes is estimated by the size of a view obtained from lattice agreement; since views are comparable, the estimate is within a constant factor (see Lemma 5.1).

In Algorithm 4, process p_i belongs to a set S_j if the size of its view is in $(2^{j-1}, 2^j]$. For views obtained in lattice agreement, this partition guarantees that $|S_j| \leq 2^j$, for $j \leq \lceil \log n \rceil$; moreover, if the number of active processes is k, then $|S_j| = 0$, for $j > \lceil \log k \rceil$. There are $\lceil \log n \rceil + 1$ copies of the Algorithm 3, denoted indRenaming$_{2^0}$, ..., indRenaming$_{2^{\lceil \log n \rceil}}$. Processes in S_j perform indRenaming$_{2^j}$, designed for 2^j processes, and obtain names in a range of size $2|S_j|-1$. The name spaces for $S_0, \ldots, S_{\lceil \log n \rceil}$ do not overlap, and their size is linear k (Figure 7).

LEMMA 5.1. *If the views of processes in a set S satisfy the comparability and self-inclusion properties of lattice agreement, and the size of a view is at most k, then $|S| \leq k$.*

Proof. Assume V is the view with maximal size in S. Let V_i be the view of some process $p_i \in S$. The self-inclusion property implies that $p_i \in V_i$, and the comparability

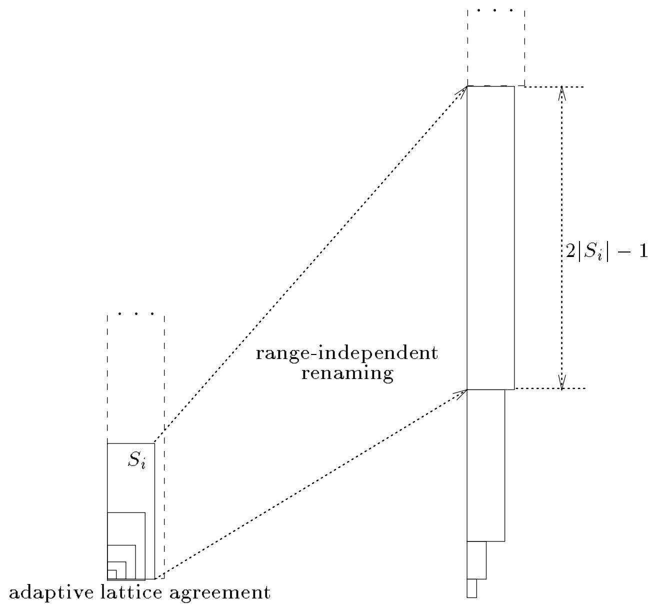

FIG. 7. *Adaptive* $(6k-1)$*-renaming.*

property implies that $V_i \subseteq V$. Therefore, $S \subseteq V$, implying that $|S| \leq |V| \leq k$. ∎

By the algorithm, if process p_i is in S_j then $|V_i| \leq 2^j$. Lemma 5.1 implies the next lemma:

LEMMA 5.2. *If there are* k *active processes, then* $|S_j| \leq 2^j$*, for* $0 \leq j \leq \lceil \log k \rceil$.

For every process p_i, $|V_i| \leq k$, since the views contain only active processes. Therefore, $p_i \in S_j$ only if $0 \leq j \leq \lceil \log k \rceil$, which implies the next lemma:

LEMMA 5.3. *If there are* k *active processes, then* $|S_j| = 0$*, for* $j > \lceil \log k \rceil$.

By Lemma 5.2, at most 2^j processes invoke $\mathsf{indRenaming}_{2^j}$. Therefore, process p_i invoking $\mathsf{indRenaming}_{2^j}$ obtains $\textit{temp-name}_i \in \{0, \ldots, 2 \cdot 2^j - 2\}$, by Theorem 4.2. By the algorithm, p_i returns $\textit{temp-name}_i + 2^{j+1} \in \{2^{j+1}, \ldots, 2^{j+2} - 2\}$.

The set of names returned by processes performing $\mathsf{indRenaming}_{2^j}$ is denoted $\textit{NameSpace}_j$; the next lemma follows from the algorithm:

LEMMA 5.4. *(1)* $\textit{NameSpace}_i \bigcap \textit{NameSpace}_j = \emptyset$*, for every* i *and* j*,* $0 \leq i < j \leq \lceil \log n \rceil$.
(2) $\bigcup_{i=0}^{m} \textit{NameSpace}_i \subseteq \{0, \ldots, 4 \cdot 2^m - 2\}$*, for every* $m \leq \lceil \log n \rceil$.

LEMMA 5.5. *If there are* k *active processes, then they return distinct names in the range* $\{0 \ldots 2^{\lceil \log k \rceil + 1} + 2k - 2\}$.[3]

Proof. If two active processes, p_i and p_j, execute the same copy of $\mathsf{indRenaming}$, then they obtain distinct names by Theorem 4.2; otherwise, they obtain distinct names, by Lemma 5.4(1).

By Lemma 5.3, processes invoke $\mathsf{indRenaming}_{2^j}$ only if $0 \leq j \leq \lceil \log k \rceil$. By Lemma 5.4(2), processes invoking $\mathsf{indRenaming}_{2^j}$, for $0 \leq j < \lceil \log k \rceil$, return names in the range $\{0, \ldots, 2^{\lceil \log k \rceil + 1} - 2\}$. By Theorem 4.2, a process p_i invoking the last non-empty copy $\mathsf{indRenaming}_{2^{\lceil \log k \rceil}}$ obtains a temporary name in the range $\{0, \ldots, 2k-2\}$. By the algorithm, p_i returns a name in the range $\{2^{\lceil \log k \rceil + 1}, \ldots, 2^{\lceil \log k \rceil + 1} + 2k - 2\}$.

[3] There are $\lceil \log k \rceil + 1$ names of the form $2^{j+2} - 1$, $0 \leq j \leq \lceil \log k \rceil$, which are not used. Therefore, the names obtained in the algorithm can be mapped into a name space of size $6k - \lceil \log k \rceil - 2$.

Thus, the output names are in a range whose size is not greater than $2^{\lceil \log k \rceil + 1} + 2k - 1$. □

Since $6k - 1 \leq 2^{\lceil \log k \rceil + 1} + 2k - 1$, the correctness of the algorithm follows from Lemma 5.5.

If there are k active processes, then each process performs AdaptiveLA (presented in the next section) in $O(k \log k)$ operations. By Lemma 5.3, only copies of indRenaming for less than $2k$ processes are invoked. Therefore, a process completes indRenaming in $O(k \log k)$ operations.

THEOREM 5.6. *Algorithm 4 solves* $(6k-1)$*-renaming with* $O(k \log k)$ *step complexity.*

The upper bound on the size of the name space, $6k - 1$, is tight for Algorithm 4. Assume that all processes executing lattice agreement obtain the maximal view (with size k) and access $\text{indRenaming}_{2^{\lceil \log k \rceil}}$. The processes leave the range $\{0, \ldots, 2^{\lceil \log k \rceil + 1} - 2\}$ unused (since it is unknown whether the previous copies of indRenaming are empty or not) and return names in the range $\{2^{\lceil \log k \rceil + 1}, \ldots, 2^{\lceil \log k \rceil + 1} + 2k - 2\}$. If k is not an integral power of 2, then the output names are in a range of size $\leq 2^{\log k + 2} + 2k - 1 = 6k - 1$. If k is an integral power of 2, then the output names are in a range of size $2^{\log k + 1} + 2k - 1 = 4k - 1$.

Merritt (private communication) noted that the names can be reduced by partitioning the active processes into sets of size $a^0, \ldots, a^j, \ldots$, for an integer $a > 2$. Active processes are partitioned into sets $S_0, \ldots, S_j, \ldots$; processes in S_j execute $\text{adaptiveRenaming}_{a^j}$ designed for a^j participants and obtain new names in a range of size $2|S_j| - 1$. As in our algorithm, when k processes are active, $\text{adaptiveRenaming}_{a^j}$ is accessed only for $0 \leq j \leq \lceil \log_a k \rceil$. Processes accessing copies $\text{adaptiveRenaming}_{a^j}$, $0 \leq j < \lceil \log_a k \rceil$, obtain names in a space of size $\sum_{j=0}^{\lceil \log_a k \rceil - 1} (2a^j - 1) \leq 2\frac{a^{\lceil \log_a k \rceil} - 1}{a-1} \leq 2\frac{a^{\log_a k + 1} - 1}{a-1} = 2\frac{ka-1}{a-1}$, which tends to $2k$, when $a \to \infty$. Processes accessing the last non-empty copy, $\text{adaptiveRenaming}_{a^{\lceil \log_a k \rceil}}$, obtain new names in a range of size $2k-1$. Thus, the size of the total name space is $\leq 4k - 1$, when $a \to \infty$.

6. Lattice Agreement in $O(k \log k)$ Operations. Our lattice agreement algorithm is based on the algorithm of Inoue et al. [23]. In their algorithm, each process starts at a distinct leaf (based on its name) of a complete binary tree with height $\lceil \log N \rceil - 1$, and climbs up the tree to the root. At each vertex on the path, it performs a procedure which merges together two sets of views, each set containing only comparable views; this procedure is called union. At the leaf, the process uses its own name as input to union; at the inner vertices, it uses the view obtained in the previous vertex as input to union. The process outputs the view it obtains at the root.

Specifically, union takes two parameters, an input view V and an integer $side \in \{0, 1\}$, and returns an output view; its properties are specified by the next lemma [23, Lemma 6]:

LEMMA 6.1. *If the input views of processes invoking* union *with* $side = 0$ *are comparable and satisfy the self-inclusion property, and similarly for the input views of processes invoking* union *with* $side = 1$*, then:*
(1) the output views of processes exiting union *are comparable, and*
(2) the output view of a process exiting union *contains its input view.*

Appendix A describes union in detail, and explains the next lemma:

LEMMA 6.2. *The step complexity of* union *is* $O(k)$.

Our adaptive algorithm uses an unbalanced binary tree T_r defined inductively as follows. T_0 has a root v_0 with a single left child (Figure 8(a)). For $r \geq 0$, suppose T_r

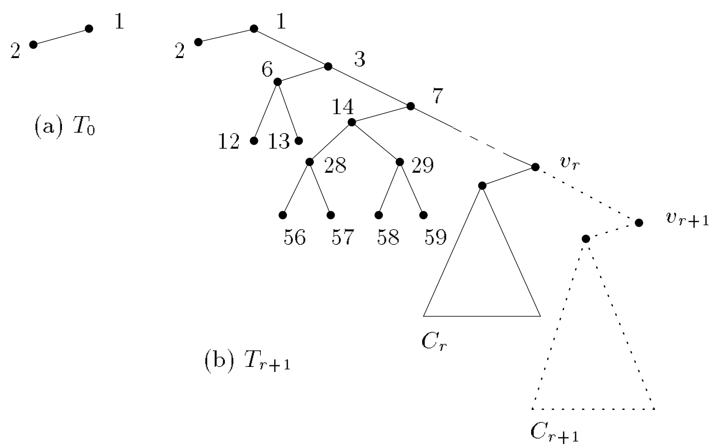

FIG. 8. *The unbalanced binary tree used in the adaptive lattice agreement algorithm.*

is defined with an identified vertex v_r, which is the last vertex in an in-order traversal of T_r; notice that v_r does not have a right child in T_r. T_{r+1} is obtained by inserting a new vertex v_{r+1} as the right child of v_r, and inserting a complete binary tree C_{r+1} of height $r+1$ as the left subtree of v_{r+1} (Figure 8(b)). By the construction, v_{r+1} is the last vertex in an in-order traversal of T_{r+1}.

The vertices of the tree are numbered as follows: The root is numbered 1; if a vertex is numbered ℓ, then its left child is numbered 2ℓ, and its right child is numbered $2\ell+1$ (Figure 8).

By the construction, the leaves of T_r are the leaves of the complete binary subtrees $C_0, C_1, \ldots, C_r$. Therefore, the total number of leaves in T_r is $\sum_{j=0}^{r} 2^j = 2^{r+1}-1$. The following simple lemma, proved in Appendix B, states some properties of T_r.

LEMMA 6.3. *Let w be the i-th leaf of T_r, $1 \leq i \leq 2^{r+1}-1$, counting from left to right. Then*
(1) the depth of w is $2\lfloor \log i \rfloor + 1$;
(2) w is numbered $2^d(2^{d+2}-3)+i$, where $d = \lfloor \log i \rfloor$.

Algorithm 5 uses $T_{2\log n-1}$, which has n^2-1 leaves.[4] A process starts the algorithm by obtaining a new name in a range of size $k(k+1)/2$ (using Algorithm 1). This name determines the leaf at which the process starts to climb up the tree: A process with a new name x_i starts the algorithm at the x_ith leaf of the tree, counting from left to right. Since $k(k+1)/2 \leq n^2-1$ for all $n \geq 2$, $T_{2\log n-1}$ have enough leaves for temporary names in a range of size $k(k+1)/2$. By Lemma 6.3, the x_ith leaf is numbered $2^d(2^{d+2}-3)+x_i$, where $d = \lfloor \log x_i \rfloor$.

As in [23], a distinct copy of union is associated with each inner vertex of the tree. A process performs copies of union associated with the vertices along its path to the root, and returns the view obtained at the root.

Simple induction on the distance of a vertex v from the leaves shows that the views obtained by processes executing union at v satisfy the comparability and self-inclusion properties. In the base case, v is a leaf and the claim is trivial since a single process starts at each leaf; the induction step follows immediately from Lemma 6.1. Hence, the views obtained at the root have the lattice agreement properties.

If there are k active processes, process p_i gets a unique name $x_i \in \{1, \ldots, k(k+$

[4]For simplicity, we assume n is a power of 2.

Algorithm 5 Adaptive lattice agreement.

Procedure AdaptiveLA()
1. $\textit{temp-name} := \mathsf{Adaptive\_}k(k+1)/2\text{-}\mathsf{renaming}()+1$ // Algorithm 1
2. $d := \lfloor \log \textit{temp-name} \rfloor$
3. $\ell := 2^d(2^{d+2} - 3) + \textit{temp-name}$ // the leaf corresponding to *temp-name*
4. $V := \{p_i\}$ // the input is the process's name
5. while ($\ell \geq 1$) do
6. $\quad$ $\textit{side} := \ell \bmod 2$ // calculate side
7. $\quad$ $\ell := \lfloor \ell/2 \rfloor$ // calculate father
8. $\quad$ $V := \mathsf{union}[\ell](V, \textit{side})$
9. return(V)

$1)/2\}$ (Line 1) and by Lemma 6.3(2), starts in a leaf ℓ of depth $2\lfloor \log x_i \rfloor + 1$ (Line 3). Therefore, p_i accesses at most $2\lfloor \log x_i \rfloor + 1 \leq 2\lfloor \log k(k+1)/2 \rfloor + 1 \leq 4 \log k + 1$ vertices. At each vertex, the execution of union requires $O(k)$ operations (Lemma 6.2). Thus, the total step complexity of the algorithm is $O(k \log k)$, implying the following theorem:

THEOREM 6.4. *Algorithm 5 solves lattice agreement with $O(k \log k)$ step complexity.*

7. Discussion. This work presents *adaptive* wait-free algorithms, whose step complexity depends only on the number of active processes, for lattice agreement and $(6k-1)$-renaming in the read/write asynchronous shared-memory model; the step complexity of both algorithms is $O(k \log k)$.

Clearly, the complexities of our algorithms—the number of steps, the number and the size of registers used—can be improved. For example, an algorithm for $O(k)$-renaming with $O(k)$ step complexity would immediately yield a lattice agreement algorithm with the same step complexity. Also it would be interesting to see if ideas from our efficient algorithms can improve the complexities of algorithms which adapt to the current contention [2, 6].

Acknowledgments:. We thank Yehuda Afek and Eli Gafni for helpful discussions, Yossi Levanoni for comments on an earlier version of the paper, and the reviewers for many suggestions on how to improve the organization and presentation.

REFERENCES

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, *Atomic snapshots of shared memory*, J. Assoc. Comput. Mach., 40 (1993), pp. 873–890.
[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., 1999, pp. 91–103.
[3] Y. Afek, D. Dauber, and D. Touitou, *Wait-free made fast*, in Proc. 27th ACM Symp. Theory of Comp., 1995, pp. 538–547.
[4] Y. Afek and M. Merritt, *Fast, wait-free $(2k-1)$-renaming*, in Proc. 18th ACM Symp. Principles of Dist. Comp., 1999, pp. 105–112.
[5] Y. Afek, G. Stupp, and D. Touitou, *Long-lived adaptive splitter and applications*. Unpublished manuscript, Dec. 1999.
[6] ———, *Long-lived and adaptive collect with applications*, in Proc. 40th IEEE Symp. Foundations of Comp. Sci., 1999, pp. 262–272.
[7] ———, *Long-lived and adaptive atomic snapshot and immediate snapshot*, in Proc. 19th ACM Symp. Principles of Dist. Comp., 2000, pp. 71–80.
[8] R. Alur and G. Taubenfeld, *Results about fast mutual exclusion*, in Proceedings of the Real-Time Systems Symposium, Dec. 1992, pp. 12–22.

[9] J. H. Anderson and M. Moir, *Using local-spin k-exclusion algorithms to improve wait-free object implementation*, Dist. Comp., 11 (1997), pp. 1–20.
[10] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk, *Renaming in an asynchronous environment*, J. Assoc. Comput. Mach., 37 (1990), pp. 524–548.
[11] H. Attiya and V. Bortnikov, *Adaptive and efficient mutual exclusion*, in Proc. 19th ACM Symp. Principles of Dist. Comp., 2000, pp. 91–100.
[12] H. Attiya and A. Fouren, *Adaptive wait-free algorithms for lattice agreement and renaming*, in Proc. 17th ACM Symp. Principles of Dist. Comp., 1998, pp. 277–286.
[13] ———, *An adaptive collect algorithm with applications.* Submitted for publication. Available at www.cs.technion.ac.il/~hagit/pubs/AF99ful.ps.gz, Aug. 1999.
[14] ———, *Polynomial and adaptive long-lived (2k-1)-renaming*, in Proc. 14th Int. Conf. on Dist. Computing, 2000, pp. 149–163.
[15] H. Attiya, M. Herlihy, and O. Rachman, *Atomic snapshots using lattice agreement*, Dist. Comp., 8 (1995), pp. 121–132.
[16] H. Attiya and O. Rachman, *Atomic snapshots in $O(n \log n)$ operations*, SIAM J. Comput., 27 (1998), pp. 319–340.
[17] 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 (1993), pp. 21–39.
[18] E. Borowsky and E. Gafni, *Immediate atomic snapshots and fast renaming*, in Proc. 12th ACM Symp. Principles of Dist. Comp., 1993, pp. 41–52.
[19] J. E. Burns and G. L. Peterson, *The ambiguity of choosing*, in Proc. 8th ACM Symp. Principles of Dist. Comp., 1989, pp. 145–158.
[20] M. Choy and A. K. Singh, *Adaptive solutions to the mutual exclusion problem*, Dist. Comp., 8 (1994), pp. 1–17.
[21] A. Fouren, *Exponential examples for two renaming algorithms.* Available at www.cs.technion.ac.il/~hagit/pubs/expo.ps.gz, Aug. 1999.
[22] M. Herlihy and N. Shavit, *The topological structure of asynchronous computability*, J. Assoc. Comput. Mach., 46 (1999), pp. 858–923.
[23] M. Inoue, W. Chen, T. Masuzawa, and N. Tokura, *Linear-time snapshot using multi-writer multi-reader registers*, in Proc. 8th Int. Workshop on Dist. Algorithms, 1994.
[24] L. Lamport, *A new solution of Dijkstra's concurrent programming problem*, Comm. ACM, 18 (1974), pp. 453–455.
[25] ———, *A fast mutual exclusion algorithm*, ACM Trans. Comput. Syst., 5 (1987), pp. 1–11.
[26] M. Moir, *Fast, long-lived renaming improved and simplified*, Sci. Comput. Programming, 30 (1998), pp. 287–308.
[27] M. Moir and J. H. Anderson, *Wait-free algorithms for fast, long-lived renaming*, Sci. Comput. Programming, 25 (1995), pp. 1–39.
[28] M. Moir and J. A. Garay, *Fast long-lived renaming improved and simplified*, in Proc. 10th Int. Workshop on Dist. Algorithms, vol. 1151 of Lecture Notes in Computer Science, Springer-Verlag, 1996, pp. 287–303.

Appendix A. Procedure union.

Procedure union (from [23]) takes two parameters, a view V and an integer $side \in \{0,1\}$, and returns a view. Let S_b be the set of processes which call union with side b. If the input views of the processes in S_0 are comparable and the input views of the processes in S_1 are comparable (within each set separately), then *all* the views returned by union are comparable. In addition, the output view of each process contains its input view.

The procedure uses two shared arrays, $Views_0[1,\ldots,n+1]$ and $Views_1[1,\ldots,n+1]$, whose entries contain views (initially empty). Processes use two procedures: WriteSet which writes a view into an array, and ReadSet which reads the view of maximal size written in an array.

A process invoking union with side b, first writes its input view V into $Views_b$ using WriteSet. Then the process repeatedly reads from the arrays $Views_0$ and $Views_1$ using ReadSet. If the last two reads from $Views_0$ return the same view, and the last two reads from $Views_1$ return the same view, then the process returns the union of these views. As in Afek et al. [1], if two non-overlapping ReadSet operations return the same maximal view, then the maximal view does not change in between the operations. Since reads from $Views_0$ alternate with reads from $Views_1$, there is an execution interval during which $Views_0$ and $Views_1$ do not change. Therefore, the union of the last views read from the arrays is a snapshot of the maximal size views written in them. If the views written in each array are comparable, then the snapshots of the maximal views, returned by union, are also comparable.

The crux of the algorithm is how ReadSet and WriteSet guarantee linear step complexity. For $b = 0, 1$, the ℓ'th entry of $Views_b$ corresponds to the unique view of size ℓ in S_b. A process accesses each entry of $Views_b$ at most once. Towards this end, if a process reads a view V in ReadSet from an entry $i < |V|$, then it writes V into entries $i+1, \ldots, |V|$; this guarantees that V will not be overwritten by a smaller view and will be available for other processes. Furthermore, if the maximal size of views in S_b is k, then a process accesses only the first $k+1$ entries of $Views_b$. Thus, a process performs at most $O(|S_0 \bigcup S_1|)$ reads and writes.

The pseudocode appears in Algorithm 6.

In WriteSet(A, V), a process writes V to $A[1], A[2], \ldots, A[|V|]$ (in this order). When there are k active processes, $|V| \leq k$, and hence no process writes to $A[k+1]$.

In ReadSet(A, V), process p_i reads the entries of array A in increasing order of indices, until it reads $\emptyset$; by the property of WriteSet, p_i obtains $\emptyset$ after at most $k+1$ reads. If p_i reads a view W from $A[j]$, $j \leq |W|$, then p_i writes W to the array in the order $A[j+1], A[j+2], \ldots, A[|W|]$ (Lines 4-6) and resumes reading from $A[|W|+1]$ (Lines 7-11). This requires ReadSet to take an additional parameter V, the maximal view p_i has written or read so far.

Let V_i be the view returned by process p_i. Clearly, the step complexity of union is linear in $|V_i|$ (see [23, Lemma 9]). Since $V_i \subseteq S_0 \cup S_1$, the step complexity of union is $O(|S_0 \cup S_1|) = O(k)$, which implies Lemma 6.2.

Appendix B. Proof of Lemma 6.3.

(1) Suppose that w belongs to a C_d, $d \leq r$. The total number of leaves in $C_0, \ldots, C_{d-1}$ is $\sum_{j=0}^{d-1} 2^j = 2^d - 1$. Therefore, $2^d - 1 < i$. Since w belongs to C_d, $i \leq \sum_{j=0}^{d} 2^j = 2^{d+1} - 1$. Thus, $2^d \leq i < 2^{d+1}$, implying that $d = \lfloor \log i \rfloor$.

Consider the path from the root of T_r to w. The path has length $2d+1$: d edges on the path from the root of T_r to v_d, one edge connecting v_d and the root of C_d, and d edges on the path from the root of C_d to w. Since $d = \lfloor \log w_i \rfloor$, the length of the path is $2\lfloor \log i \rfloor + 1$.

(2) Define a binary string $b[1 \ldots 2d+1]$, which represents the path from the root of T_r to w, as follows: $b[1] = 1$; for any $1 \leq j < 2d+1$, $b[j+1] = 0$, if the $(j+1)$th vertex on the path is the left child of its parent; and $b[j+1] = 1$, otherwise.

The numbering of the vertices in T_r implies that the number of w is represented by the binary string $b[1 \ldots 2d+1]$. The d leftmost bits of b, corresponding to the path from the root of the tree to vertex v_d are $11\ldots1$, since all the vertices on the path are right children (see Figure 8(b)). The $(d+1)$th bit of b is 0, since the root of C_d is the left child of v_d. It remains to calculate the d rightmost bits of b, corresponding to the path from the root of C_d to w.

The total number of leaves in $C_0, \ldots, C_{d-1}$ is $\sum_{j=0}^{d-1} 2^j = 2^d - 1$. Since w is the ith leaf in T_r, it is the $(i - (2^d - 1))$th leaf in C_d, counting from 1. Therefore, the d bits corresponding to the path from the root of C_d to w are the binary representation of the number $(i - (2^d - 1)) - 1$, that is, $(i - 2^d)_2$. Hence, $b[1 \ldots 2d+1] = \overbrace{11\ldots1}^{d}0 \cdot (i - 2^d)_2$.

Algorithm 6 Procedure union (from [23]).

Procedure union(V: view, b: 0,1) // side parameter is b
WriteSet($Views_b$,V)
$current_b := V$; $current_{1-b} := \emptyset$
repeat
  $prev_0 := current_0$; $prev_1 := current_1$
  $current_0 :=$ ReadSet($Views_0$,$prev_0$)
  $current_1 :=$ ReadSet($Views_1$,$prev_1$)
until ($prev_0 = current_0$ and $prev_1 = current_1$)
return($current_0 \cup current_1$)

Procedure WriteSet(A: array of registers, V: view)
for $j = 1$ to $|V|$ do $A[j] := V$

Procedure ReadSet(A: array of registers, $prev$: view)
$max := prev$
$p := |max|$
while ($p \leq |max|$) do
  for $j = p + 1$ to $|max|$ do $A[j] := max$
  $p := |max|$
  repeat
    $p := p + 1$
    $temp := A[p]$
    if ($|temp| > p$) then $max := temp$
  until ($|max| > p$ or $temp = \emptyset$)
return(max)

Therefore, the number of w is $(2^{d+1} - 1)2^{d+1} + i - 2^d = 2^d(2^{d+2} - 3) + i$.

Appendix C. Linear Atomic Snapshots.

This section presents a linear-time algorithm for lattice agreement using only dynamic single-writer single-reader registers; this algorithm is neither range-independent nor adaptive. It appears here because it is a simple modification of Algorithm 2, and it shows yet another interesting connection between renaming and lattice agreement.

The algorithm uses the network of reflectors of Figure 3. In the modified algorithm, reflectors help the processes to collect views, in addition to directing their movements. The modified reflector is implemented with two N-bit registers, R_0 and R_1, that can contain views. Initially, each register contains the empty view, $\emptyset$. A process entering the reflector on entrance in_i writes its local view into register R_i, and then reads the other register, R_{1-i}. If it is $\emptyset$, then the local view of the process does not change, and the process exits on $down_i$; otherwise, the process joins the view written in R_{1-i} with its local view and exits on up_i.

The algorithm which controls the processes' movements in the network remains exactly the same as in Algorithm 2. Processes leaving the network return their local views (instead of the row numbers, as in the renaming algorithm). Pseudocode appears in Algorithm 7; the numbers of the modified lines appear in bold.

For a process p_i and a reflector S, $V_{in}(p_i, S)$ and $V_{out}(p_i, S)$ are the views of of p_i before and after accessing S, respectively. The following lemma is a modification of Lemma 3.1. The only difference is related to the process's view after accessing a

Algorithm 7 Linear lattice agreement.

```
Procedure LatticeAgreement(name_i : 0, ..., N − 1)
private col, row: integer, initially name_i   // start on top reflector of column name_i
private V, initially {p_i}                                        // the initial view
while ( row = name_i ) do                           // descend through column name_i
    ⟨V, exit⟩ := reflector[row,col](V,1)                              // enter on in_1
    if ( exit = up_1 ) then col++;
    else                                                           // exit = down_1
        row−−;
        if ( row < −col ) then
            col++;                        // reached the lowest reflector in column
while ( col < N ) do                                // move towards column N − 1
    ⟨V, exit⟩ := reflector[row,col](V,0)                              // enter on in_1
    if ( exit = up_0 ) then col++; row++;
    else col++; row−−;                                             // exit = down_0
return(V);

Procedure reflector(view V, entrance r: 0,1)
R_r := V;
if ( R_{1−r} = ∅ ) then return(⟨V, down_r⟩)
else return(⟨V ∪ R_{1−r}, up_r⟩)
```

reflector. By the algorithm, the local view of the process which reads $\emptyset$ from R_{1-i} does not change.

LEMMA C.1. *If a single process p_j enters a reflector S then p_j leaves S on a lower exit and $V_{out}(p_i, S) = V_{in}(p_i, S)$.*

The proof of the following lemma is similar to Lemma 3.2.

LEMMA C.2. *If a single process p_i enters a reflector S on in_0, and a single process p_j enters S on in_1, then:*
(1) at most one of the processes leaves S on a lower exit;
(2) if $p \in \{p_i, p_j\}$ goes down, then $V_{out}(p, S) = V_{in}(p, S)$;
(3) if $p \in \{p_i, p_j\}$ goes up, then $V_{out}(p, S) = V_{in}(p_i, S) \cup V_{in}(p_j, S)$.

Recall that S_c is the set of processes which start execution on columns $0, \dots, c$; let $V(p_i, c)$ be the view with which a process p_i exists column c. The next lemma shows that the views of processes exiting column c are comparable.

LEMMA C.3. *For every pair of processes $p_i, p_j \in S_c$, $0 \leq c < N$, if $row(p_i, c) < row(p_j, c)$ then $V(p_i, c) \subseteq V(p_j, c)$.*

Proof. The proof is by induction on the column c. The lemma holds in the base case, $c = 0$, since at most one process exits column 0. Suppose that the lemma holds for $c \geq 0$.

If there is no active process with name equal to $c+1$, then the reflectors in column $c+1$ are accessed only on entrances in_0. Therefore, by Lemma C.1, the local views of the processes S_c do not change in column $c+1$, and for every process $p_j \in S_c$, $row(p_j, c+1) = row(p_j, c) - 1$. In this case, the lemma holds by the induction hypothesis.

Otherwise, let q be an active process which starts on column $c+1$. Suppose that the last reflector accessed by q on column $c+1$ is $S[c+1, r']$, $-(c+1) \leq r' \leq c+1$. That is, q leaves reflectors $S[c+1, c+1], \dots, S[c+1, r'+1]$ on $down_1$, and q does not access reflectors $S[c+1, r'-1], \ldots, S[c+1, -(c+1)]$. By Lemmas C.1 and C.2:

– for all $p_j \in S_c$ such that $row(p_i, c) < r'$

$$\begin{aligned} exit(p_i, c+1) &= down_0 \\ row(p_i, c+1) &= row(p_i, c) - 1 \\ V(p_i, c+1) &= V(p_i, c) \end{aligned} \tag{4}$$

– for all $p_j \in S_c$ such that $row(p_i, c) > r'$

$$\begin{aligned} exit(p_i, c+1) &= up_0 \\ row(p_i, c+1) &= row(p_i, c) + 1 \\ V(p_i, c+1) &= V(p_i, c) \cup \{q\} \end{aligned} \tag{5}$$

By the algorithm, q leaves column $c+1$ either on exit $down_1$ of the lowest reflector $S[c+1, -(c+1)]$ in the column, or on exit up_1 of $S[c+1, r']$, where $-(c+1) \leq r' \leq c+1$.

If q leaves reflector $S[c+1, -(c+1)]$ on $down_1$, then by Lemma C.1, $row(q, c+1) = -(c+1) - 1$ and $V(q, c+1) = V(q, c)$. The lemma holds by the induction hypothesis and (5).

If q leaves reflector $S[c+1, r']$ on up_1, then there is a process $p_j \in S_c$ such that $row(p_j, c) = r'$, by Lemma C.1. By Lemma C.2, $V(q, c+1) = V(p_j, c) \cup \{q\}$. If p_j leaves column $c+1$ on up_0, then $V(p_j, c+1) = V(p_j, c) \cup \{q\}$ and $row(p_j, c+1) = row(p_j, c) + 1 = r' + 1$, by Lemma C.2. If p_j leaves column $c+1$ on $down_0$, then $V(p_j, c+1) = V(p_j, c)$ and $row(p_j, c+1) = r' - 1$, by Lemma C.2. In both cases, the lemma holds by the induction hypothesis, (4) and (5). □

Lemma C.3 with $c = N - 1$ implies that the local views of the processes leaving the network are comparable. Since p_i initial view contains its identifier and the local view never decreases, we have the self-inclusion property. The step complexity of the algorithm is calculated as in Theorem 3.5. The network contains $O(N^2)$ reflectors; each reflector uses two bounded dynamic single-writer single-reader registers. This proves the following theorem:

THEOREM C.4. *Algorithm 7 solves the lattice agreement problem with step complexity $O(N)$ using $O(N^2)$ dynamic single-writer single-reader registers of length N bits.*

Applying the transformation of [15], which requires $O(N)$ additional operations on single-writer multi-reader registers, we get the following result:

THEOREM C.5. *There is an implementation of an atomic snapshot object such that every operation on the object requires $O(N)$ read/write operations on dynamic single-writer multi-reader registers. The implementation uses $O(N^2)$ registers of length N bits.*