Algorithm 9 Procedure union: code for process \( p_i \) (from Inoue, Chen, Masuzawa and Tokura [15])

Procedure union(V: view, side: 0,1)
var \( L[1,\ldots,n+1] \), \( R[1,\ldots,n+1] \) : array of registers, each register is initialized to \( \emptyset \)
1. if(side=0)
2. then
3. WriteSet(L, V)
4. \( prev_l := V \)
5. \( prev_r := \emptyset \)
6. else
7. WriteSet(R, V)
8. \( prev_l := \emptyset \)
9. \( prev_r := V \)
10. \( current_l := \text{ReadSet}(L, prev_l) \)
11. \( current_r := \text{ReadSet}(R, prev_r) \)
12. while (\( prev_l \neq current_l \) or \( prev_r \neq current_r \))
13. \( prev_l := current_l \)
14. \( prev_r := current_r \)
15. \( current_l := \text{ReadSet}(L, prev_l) \)
16. \( current_r := \text{ReadSet}(R, prev_r) \)
17. return (\( current_l \cup current_r \) )

Procedure WriteSet(A: array of registers, V: view)
1. for \( j=1 \to |V| \)
2. write(A[j], V)

Procedure ReadSet(A: array of registers, prev: view)
1. \( max := prev \)
2. \( var_p := |max| \)
3. while (\( p \leq |max| \))
4. for \( j = p+1 \) to \(|max| \)
5. write(A[j], max)
6. \( p := |max| \)
7. repeat
8. \( p := p+1 \)
9. \( temp := \text{read}(A[p]) \)
10. if (\( |temp| > p \)) then \( max := temp \)
11. until (\( |max| > p \) or \( temp = \emptyset \))
12. return(max)
A process which enters `union` on entrance \( in_0 \) starts by writing its input view \( V \) into array \( L \) using `WriteSet`. Similarly, a process which enters `union` on \( in_1 \), writes its input view \( V \) into array \( R \). Then the process repeatedly reads \( L \) and \( R \) using `ReadSet`. If the last two reads from \( L \) return the same view, and the last two reads from \( R \) return the same view, then the process decides on the union of these views.

The algorithm uses the same technique of repeated reading, as the one used in the \( O(k^2) \) lattice agreement (Algorithm 2). By the similar arguments, if two non-overlapping operations `ReadSet` return the same maximal view written in an array, then the maximal view does not change in between the operations. Since reads from \( L \) alternate with reads from \( R \), there is a time interval on which both arrays, \( L \) and \( R \) do not change. Therefore, the union of the last views read from \( L \) and \( R \) constitutes a snapshot of the maximal size views written in the both arrays. If the views written in each array satisfy the inclusion relation, then the snapshots of the maximal views, returned by `union`, also satisfy the inclusion relation.

We now describe execution of `WriteSet` and `ReadSet` by a process \( p_i \).

In `WriteSet(A,V)` which writes a view \( V \) into an array \( A \), \( p_i \) writes \( V \) into the registers in the order \( A[1], A[2], \ldots, A[|V|] \).

In `ReadSet(A)` which reads a view from an array \( A \), \( p_i \) reads the registers in the ascending order of their indices, until it reads \( \emptyset \). Since the size of a view is not greater then \( n \), no process writes a view to \( A[n+1] \). Thus, \( p_i \) is guaranteed to get \( \emptyset \) after at most \( n + 1 \) reads.

In `ReadSet(A)` \( p_i \) never reads the register to which \( p_i \) has written or from which \( p_i \) has read a non-empty set until then. During the execution, if \( p_i \) reads a view \( W \) from the register whose index \( j \) is less then the size of \( W \), \( p_i \) writes \( W \) to the registers in the order \( A[j+1], A[j+2], \ldots, A[|W|] \), and then resumes to read registers from \( A[|W|+1] \).

For any process \( p \), let \( V_{in}(p) \) and \( V_{out}(p) \) be the input parameter and the returned view, respectively. The following lemma, from [15], states the main properties of `union`:

**Lemma A.1** If the input views of the processes in \( S_0 \) satisfy the inclusion relation, and the input views of the processes in \( S_1 \) satisfy the inclusion relation, then the following conditions hold:

1. the views of the processes exiting `union` satisfy the inclusion relation
2. the output of a process exiting `union` includes its input view.

As shown in [15], step complexity of procedure `union` is linear in the maximal size of returned view. For any process \( p_i \), \( V_{out}(p_i) \subseteq S_0 \cup S_1 \). Therefore, procedure `union` has step complexity \( O(|S_0 \cup S_1|) = O(k) \). Procedure `union` does not use \textit{a-priori} knowledge of \( k \), therefore the algorithm is adaptive. Thus we have:

**Lemma A.2** Procedure `union` shown in Algorithm 9 has adaptive complexity \( O(k) \).


A The union Procedure

Procedure union described here was suggested by Inoue, Chen, Masuzawa and Tokura [15] as a building block for a linear time lattice agreement algorithm.

The procedure takes two parameters, a view $V$ and an integer $\text{side} \in \{0, 1\}$, and returns a view. Let $S_i$ be the set of processes which call union with parameter $\text{side}=i$; we say also that they enter union on entrance in $S_i$. It is guaranteed, that if the input parameters $V$ of the processes in $S_0$ and in $S_1$ satisfy the inclusion relation, then the views returned by union also satisfy the inclusion relation.

We now outline the procedure. The details appear in Algorithm 9.

The procedure uses two register arrays, $L[1 \ldots n+1]$ and $R[1 \ldots n+1]$. Each register in the arrays is intended to contain a view. Initially, the registers contain empty views ($\emptyset$). Processes manipulate $L$ and $R$ using two procedures, WriteSet and ReadSet. WriteSet($A$, $V$) writes view $V$ into array $A$. ReadSet($A$) reads a view of the maximal size written to array $A$.
References


different partitioning and merging procedures, in order to optimize the overall complexity of the algorithms.

In the adaptive $O(k \log k)$ algorithm for lattice agreement, processes are partitioned into subsets according to the names obtained in an adaptive $k(k + 1)/2$-renaming. Partial results are merged using an adaptive union procedure [15].

For adaptive renaming, we did not find any efficient way to merge the partial solutions without increasing the size of the name space. The best solution we found was to choose partial name spaces that do not overlap. This forced us to use a more “compact” partitioning, based on adaptive lattice agreement instead of $O(k^2)$-renaming.

The proposed methodology may be used to transform existing non-adaptive wait-free algorithms into adaptive ones. Assume that we wish to take a wait-free algorithm $A$ for a decision problem $P$ and make it adaptive. Assume that step complexity of $A$ depends on the size of the initial name space. Then we need to design only the merging procedure, which depends on the specific problem. For the partitioning, we can use one of the techniques described in this work (based on adaptive renaming or on adaptive lattice agreement), and a partial solution for each subset can be obtained using the given non-adaptive algorithm.

### 9.2 Further Research

Our work leaves many avenues for further research.

The adaptive algorithm for renaming with linear name space is composed of a sequence of algorithms for lattice agreement and renaming with larger names space. It would be interesting to find a more direct algorithm, thereby simplifying it and, possibly, improving the constant factors. There is also room for reducing the complexities of the algorithms, either the number of steps or the number and size of registers used.

The most interesting direction for further research is to find adaptive algorithms for on-going problems, for example, long-lived renaming [17]. In this case, it is better to find adaptive algorithms that dynamically adapt to the number of processes currently active in the algorithm. The algorithms of Afek, Dauber and Touitou [2] and of Choy and Singh [11] have this property, but they either use operations stronger than reads and writes or are not wait-free.

One direction which may lead to creation of adaptive algorithms for on-going problems is the development of a procedure for dynamic estimation of the number of active processes. Using this procedure, processes can be partitioned into dynamic sets with fixed maximal size. Then in each subset processes can perform existing fast on-going algorithms (for example, fast long-lived renaming [10, 18]).

Another possibility is to develop an adaptive analog of the fast long-lived splitter of Moir and Garay [18]. An adaptive long-lived splitter will immediately lead to adaptive algorithms for long-lived collect and and renaming, using constructions similar to those presented in this work.
Figure 13: Relations between algorithms presented in the work
Lemma 8.3 with \( c = N - 1 \) implies that the local views of the processes that leave the network are comparable. Since \( V(p_i) = \{p_i\} \) initially, and the local view never decreases, we have the self-containment property. The step complexity of the algorithm is calculated as in Theorem 5.5. The network contains \( O(N^3) \) reflectors; each reflector uses two bounded dynamic single-writer single-reader registers. This proves the following theorem:

**Theorem 8.4** Algorithm 8 solves the lattice agreement problem with step complexity \( O(N) \) using \( O(N^3) \) dynamic single-writer single-reader registers of length \( N \).

If the initial name space is \( \{0, \ldots, n - 1\} \), then the complexity of Algorithm 8 is \( O(n) \). Applying the transformation of [6], which requires \( O(n) \) additional operations on single-writer multi-reader registers, we get the following result:

**Theorem 8.5** There is an implementation of atomic snapshot object such that any operation on the object requires \( O(n) \) read and write operations on dynamic single-writer multi-reader registers. The implementation uses \( O(n^2) \) registers of length \( n \).

## 9 Discussion

### 9.1 Overview of the Results

This work focuses on adaptive wait-free algorithms, whose step complexity depends only on the number of active processes, which is not known in advance. We have shown adaptive algorithms for collect, lattice agreement and renaming in the read/write asynchronous shared-memory model. The sequence of the constructions leading to these results is shown schematically in Figure 13.

One of the contributions of the work is the methodology used in the adaptive algorithms for lattice agreement and renaming. In both algorithms, active processes are first partitioned into disjoint subsets of exponentially growing size. In each subset processes solve the problem that depends only on the size of the subset. Since the maximal size of each subset is known in advance, the name space associated with a subset is also known. Thus, in each subset processes can perform algorithms relying on the knowledge of \( K \) and \( N \), in order to solve the problem inside the subset. Finally, these partial solutions are merged to form a correct solution with respect to the whole set of processes.

Thus, the methodology requires design of three main components: an adaptive partitioning procedure, a (non-adaptive) algorithm for solving the problem in each subset, and an adaptive merging procedure.

Using this methodology, we constructed an \( O(k \log k) \) adaptive algorithm for lattice agreement and an \( O(k \log k) \) adaptive algorithm for \((6k - 1)\)-renaming. In each case we choose
**Lemma 8.3** For every column \( c = 0, \ldots, N - 1 \), and for every pair of processes \( p_i \) and \( p_j \) in \( S_c \), if \( \text{row}(p_i, c) < \text{row}(p_j, c) \) then \( V(p_i, c) \subseteq V(p_j, c) \).

**Proof:** The proof is by induction on the column \( c \). In the base case, \( c = 0 \), the lemma is true, since only one process may exit column 0. Suppose that the lemma holds for \( c \geq 0 \). We will show that it holds also for column \( c + 1 \).

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 \( S_c \). Therefore, by Lemma 8.1, the local views of the processes \( S_c \) do not change in column \( c + 1 \), and for any process \( p_j \in S_c \), \( \text{row}(p_j, c + 1) = \text{row}(p_j, c) - 1 \). The lemma follows from the inductive hypothesis.

Otherwise, let \( q \) be an active process which starts at column \( c + 1 \).

Suppose that the last reflector accessed by \( q \) at column \( c + 1 \) is \( S_c \)[\( c + 1, r' \)], \(-(c + 1) \leq r' \leq c + 1 \). That is, \( q \) leaves reflectors \( S_c \)[\( c + 1, c + 1 \)], \ldots, \( S_c \)[\( c + 1, r' + 1 \)] through \( \text{down}_1 \), and \( q \) does not access reflectors \( S_c \)[\( c + 1, r' - 1 \)], \ldots, \( S_c \)[\( c + 1, -(c + 1) \)]. Therefore, by Lemmas 8.1 and 8.2 we have:

- for all \( p_j \in S_c \) such that \( \text{row}(p_i, c) < r' \)
  \[
  \begin{align*}
  \text{exit}(p_i, c + 1) &= \text{down}_0 \\
  \text{row}(p_i, c + 1) &= \text{row}(p_i, c) - 1 \\
  V(p_i, c + 1) &= V(p_i, c)
  \end{align*}
  \] (4)

- for all \( p_j \in S_c \) such that \( \text{row}(p_i, c) > r' \)
  \[
  \begin{align*}
  \text{exit}(p_i, c + 1) &= \text{up}_0 \\
  \text{row}(p_i, c + 1) &= \text{row}(p_i, c) + 1 \\
  V(p_i, c + 1) &= V(p_i, c) \cup \{q\}
  \end{align*}
  \] (5)

By the algorithm, \( q \) can leave column \( c + 1 \) through exit \( \text{down}_1 \) of the lowest reflector \( S_c \)[\( c + 1, -(c + 1) \)] in the column, or through exit \( \text{up}_1 \) of \( S_c \)[\( c + 1, r' \)], where \(-(c + 1) \leq r' \leq c + 1 \).

If \( q \) leaves reflector \( S_c \)[\( c + 1, -(c + 1) \)] through the lower exit \( \text{down}_1 \), then by Lemma 8.1, \( \text{row}(q, c + 1) = -(c + 1) - 1 \) and \( V(q, c + 1) = V(q, c) \). The lemma follows by the inductive hypothesis and (5).

If \( q \) leaves reflector \( S_c \)[\( c + 1, r' \)] through the upper exit \( \text{up}_1 \), then there must be a process \( p_j \in S_c \) such that \( \text{row}(p_j, c) = r' \), by Lemma 8.1. Then by Lemma 8.2, \( V(q, c + 1) = V(p_j, c) \cup \{q\} \).

Otherwise, \( \text{exit}(p_j, c + 1) = \text{up}_0 \); Lemma 8.2 implies that \( V(p_j, c + 1) = V(p_j, c) \cup \{q\} \) and \( \text{row}(p_j, c + 1) = \text{row}(p_j, c) + 1 = r' + 1 \). If \( \text{exit}(p_j, c + 1) = \text{down}_0 \), then by Lemma 8.2, \( V(p_j, c + 1) = V(p_j, c) \) and \( \text{row}(p_j, c + 1) = r' - 1 \). In both cases, the lemma holds by the inductive hypothesis, (4) and (5).

\[\square\]
Algorithm 8 Linear lattice agreement.

Procedure net\( (\text{name}_i : 0, \ldots, N - 1) \)

private variables:
\( V \): view, initially \( \{p_i\} \);
\( \text{in} \): integer, initially 1;
\( \text{col} \): integer, initially \( \text{name}_i \);
\( \text{row} \): integer, initially \( \text{name}_i \);

1. while(\( \text{in}=1 \)) \( // p_i \) descends through column \( \text{name}_i \)
2. \( \langle V, \text{exit} \rangle := \text{reflector}[\text{row}, \text{col}](V, \text{in}) \)
3. if \( \text{exit} = \text{up}_i \) then
4. \( \text{col}++; \text{in}:= 0; \)
5. else
6. \( \text{row}--; \)
7. if \( \text{row} < -\text{col} \) then
8. \( \text{col}++; \text{in}:= 0; \)
9. while( \( \text{col} < N \) ) \( // \) the lowest reflector is reached
10. \( \langle V, \text{exit} \rangle := \text{reflector}[\text{row}, \text{col}](V, \text{in}) \)
11. if \( \text{exit} = \text{up}_i \) then \( \text{col}++; \text{row}++; \)
12. else \( \text{col}++; \text{row}--; \)
13. decide(\( V \));

Procedure \text{reflector}(\text{view} V, \text{entrance} i)

1. write\( (R_i, V) \);
2. if \( (R_{i-1} = 0) \) then \( \text{exit} := \text{down}_i \)
3. else \( \text{exit} := \text{up}_i \)
4. \( V := V \cup R_{i-1} \)
5. return\( (\langle V, \text{exit} \rangle) \)
and Rachman [6], we get an $O(n)$ implementation of atomic snapshots using dynamic single-writer multi-reader registers.

Although this algorithm is neither fast nor adaptive, we present it here because it is a simple modification of Algorithm 5; this shows yet another interesting connection between renaming and lattice agreement.

The algorithm uses the same network of reflectors as Algorithm 5 (shown in Figure 8). In the modified algorithm, reflectors have an additional function: They not only direct the processes’ movements, but also help them to collect views. It is guaranteed that the views of processes leaving the network satisfy the properties of lattice agreement.

The new implementation of the reflector uses two $N$-bit registers, $R_0$ and $R_1$, intended to contain views. (Previously, the registers contained true or false.) 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 goes down; otherwise, the process joins the view written in $R_{1-i}$ with its local view and goes up.

The algorithm which controls the processes’ movements in the network remains exactly the same as in Algorithm 5. Processes leaving the network decide on their local views (instead of the row numbers, as in the renaming algorithm).

Pseudocode appears in Algorithm 8. Numbers of the modified lines are shown in bold.

The following lemma is a modification of Lemma 5.1. The only difference is related to the value of the local view after execution of the reflector. By the algorithm, the local view of the process which reads $\emptyset$ from $R_{1-i}$ does not change. Thus we have:

Lemma 8.1 If a single process $p_j$ enters a reflector then $p_j$ leaves the reflector through a lower exit and $V_{out}(p_j) = V_{in}(p_j)$.

The proof of the following lemma is similar to the proof of Lemma 5.2. The additions related to the local views follow immediately from the algorithm.

Lemma 8.2 If a single process $p_i$ enters the reflector on $in_0$, and a single process $p_j$ enters the reflector on $in_1$, then:

1. at most one of the processes leaves the reflector through a lower exit;
2. if $p \in \{p_i, p_j\}$ goes down, then $V_{out}(p) = V_{in}(p)$;
3. if $p \in \{p_i, p_j\}$ goes up, then $V_{out}(p) = V_{in}(p_i) \cup V_{in}(p_j)$.

The following lemma shows that the local views of the processes exiting column $c$ are comparable. Recall that $S_c$ is the set of processes which start execution at columns $0, \ldots, c$. 

35
Lemma 7.3  
(1) NameSpace$_i$ $\cap$ NameSpace$_j$ = $\emptyset$, for any $i$ and $j$, $0 \leq i < j \leq \lfloor \log n \rfloor$.
(2) $\bigcup_{i=0}^{n} \text{NameSpace}_i \subseteq \{0, \ldots, 4 \cdot 2^m - 2\}$, for any $m \leq \lfloor \log n \rfloor$.

Lemma 7.4 If there are $k$ active processes, then they decide on distinct names in the range
$\{0 \ldots 2^{\lfloor \log k \rfloor + 1} + 2k - 2\}$.

Proof: First, we show that the names are distinct. Let $p_i$ and $p_j$ be active processes. If
$p_i$ and $p_j$ execute the same copy of fastRenaming, the condition follows from the property of
renaming (Theorem 6.2); otherwise, the condition follows from Lemma 7.3(1).

We now compute the size of the final name space. By Lemma 7.2, active processes access
only fastRenaming$_{\lfloor \log k \rfloor}$, with $0 \leq j \leq \lfloor \log k \rfloor$. By Lemma 7.3 (2), processes accessing copies
fastRenaming$_{\langle j \rangle}$, $0 \leq j < \lfloor \log k \rfloor$, decide on the names in the range $\{0, \ldots, 2^{\lfloor \log k \rfloor + 1} - 2\}$.

By the property of $(2k - 1)$-renaming (Theorem 6.2), each process $p_i$ accessing the last
non-empty copy fastRenaming$_{\langle \lfloor \log k \rfloor \rangle}$ acquires temporary name in the range $\{0 \ldots 2k - 2\}$, and
by the algorithm, $p_i$ decides on a name in the range $\{2^{\lfloor \log k \rfloor + 1} \ldots 2^{\lfloor \log k \rfloor + 1} + 2k - 2\}$ Thus, the
size of the final name space is not greater than $2^{\lfloor \log k \rfloor + 1} + 2k - 1$.

This upper bound on the size of the name space, $6k - 1$, is tight for Algorithm 7. Assume
that all processes executing lattice agreement obtain the maximal view (with size $k$) and access
fastRenaming$_{\lfloor \log k \rfloor}$. The processes leave the range $\{0 \ldots 2^{\lfloor \log k \rfloor + 1} - 2\}$ unused (since they can
determine whether the previous copies of fastRenaming are empty or not) and decide on
names in $\{2^{\lfloor \log k \rfloor + 1} \ldots 2^{\lfloor \log k \rfloor + 1} + 2k - 2\}$. If $k$ is not an integral power of 2, then the size of
the final name space is $\leq 2^{\lfloor \log k \rfloor + 2} + 2k - 1 = 6k - 1$. Otherwise, the size of the name space is
$2^{\lfloor \log k \rfloor + 1} + 2k - 1 = 6k - 1$.

8 Linear Atomic Snapshot with Dynamic Single-Writer Single-
Reader Registers

In this section we presents a linear-time algorithm for lattice agreement using only dynamic
single-writer single-reader registers. Applying the transformation proposed by Attiya, Herlihy

$^3$There are $[\log k] + 1$ names of the form $2^{j+2} - 1$, $0 \leq j \leq [\log k]$, which are not used. Therefore, the names
obtained in the algorithm can be mapped into a name space of size $6k - [\log k] - 2$. 
Algorithm 7 Adaptive $(6k-1)$-renaming.

1. $V := \text{AdaptiveLA}()$
2. $r := \lceil \log |V| \rceil$
3. $\text{temp\_name} := \text{fastRenaming}_r()$
4. if ($r = 0$) then decide($\text{temp\_name}$)
5. else decide($\text{temp\_name} + 2^{r+1}$)

union, is used to merge the partial solutions for each subset $S_j$ to form a correct solution with respect to the whole set of processes. Unfortunately, we were not able to find an adaptive analog of union for the $(2k-1)$-renaming problem. Instead, we choose non-overlapping name spaces for the instances of fastRenaming$_j$ that together form a name space of size $6k-1$.

Second, in the lattice agreement algorithm active processes are partitioned into subsets according to their names, obtained in $k(k+1)/2$-renaming. In the adaptive renaming algorithm we are forced to associate a separate name space with each subset $S_j$. In order to get a linear final name space, we need a more precise estimation of the number of active processes than the one based on $O(k^2)$-renaming. The following lemmas show, that partitioning of the processes according to the sizes of comparable views provides an estimation within a constant factor of $k$.

Lemma 7.1 If the views of processes in a set $S$ satisfy the comparability and self-containment properties of lattice agreement, and maximal size of any view is $k$, then $|S| \leq k$.

Proof: Let $V(p_i)$ be a view with the maximal size. By the comparability property, for any process $p_j \in S$, $V(p_j) \subseteq V(p_i)$, and by the self-containment property, $p_j \in V(p_j)$. Therefore, $S \subseteq V(p_i), |S| \leq k$, since $|V(p_i)| = k$.

Lemma 7.2 If there are $k$ active processes, then:

1. $|S_j| \leq 2^i$, if $0 \leq j \leq \lceil \log k \rceil$, and
2. $|S_j| = 0$, if $j > \lceil \log k \rceil$.

Proof: (1) By the algorithm, a process $p_i$ belongs to $S_j$ if and only if $2^{j-1} < |V(p_i)| \leq 2^j$. Since the views of the processes satisfy the self-inclusion and containment properties of lattice agreement, Lemma 7.1 implies that $|S_j| \leq 2^j$.

(2) The views contain only active processes. Therefore, for any $p_i$, $|V(p_i)| \leq k$. By the algorithm, this implies that any active process $p_i$ belongs to a set $S_j, 0 \leq j \leq \lceil \log k \rceil$. Therefore for any $j > \lceil \log k \rceil$, $|S_j| = 0$.

By Lemma 7.2, the number of processes which access fastRenaming$_{2j}$ does not exceed its capacity $2^j$. Therefore, by Theorem 6.2, a process $p_i$ accessing fastRenaming$_{2j}$, acquires temp\_name($p_i$) in the range $\{0, \ldots, 2 \cdot 2^j - 2\}$. By the algorithm, $p_i$ decides on temp\_name($p_i$) + $2^j+1 \in \{2^j+1, \ldots, 2^{j+2} - 2\}$. Let NameSpace$_j$ be the set of names obtained by processes performing fastRenaming$_{2j}$. We have the following lemma:
and only if the size of its view is in \((2^{i-1}, 2^i]\). This partition guarantees that for any \(j \leq \lceil \log n \rceil\), \(|S_j| \leq 2^i\), and if the number of active processes is \(k\), then for any \(j > \lceil \log k \rceil\), \(|S_j| = 0\).

The algorithm uses \(\lceil \log n \rceil + 1\) copies of the fast \((2k - 1)\)-renaming algorithm (Algorithm 6), which are denoted \(\text{fastRenaming}_n\), \(\text{fastRenaming}_{2^n}\), \ldots, \(\text{fastRenaming}_{2^\lceil \log n \rceil}\). Each copy \(\text{fastRenaming}_{n^j}\) is designed to be executed by at most \(K = 2^i\) processes; the capacity, \(K\), is ‘hard-coded’ in the algorithm. Processes which belong to \(S_j\) perform \(\text{fastRenaming}_{n^j}\), and acquire new names from a name space of size \(2|S_j| - 1\). The name spaces for \(S_0, \ldots, S_{\lceil \log n \rceil}\) do not overlap, and the size of the final name space for \(k\) active processes is linear in \(k\) (see Figure 12).

Pseudocode appears in Algorithm 7.

Although this renaming algorithm is similar to the adaptive lattice agreement described in Subsection 4, there are two important differences.

First, in the lattice agreement algorithm (Algorithm 4), an adaptive unifying procedure,
Lemma 6.1 Processes executing net[v] acquire distinct temporary names in \(\{0, \ldots, 2k - 2\}\), for every vertex v.

Proof: The proof is by induction on \(h\), height of the vertex.

**Base case:** \(h = 0\). By the construction, the tree has \(2^h \geq K(K+1)/4\) leafs. After executing \(k(k+1)/2\)-renaming, processes get distinct names in \(\{0 \ldots K(K+1)/2 - 1\}\). Therefore, at most one process accesses \(v\) from the left son and at most one process accesses \(v\) from the right son. The process accessing \(v\) from the left son executes \(\text{net}[v]\) with temporary name 0, while the process accessing \(v\) from the right son executes \(\text{net}[v]\) with temporary name \(2K - 1 + 0 = 2K - 1\). By Theorem 5.5, since these two processes execute procedure \(\text{net}[v]\) with different temporary names in the range \(\{0, \ldots, 4K - 3\}\), the processes obtain distinct names in the range \(\{0, 1, 2\}\) in \(v\). Therefore, the lemma holds for \(h = 0\).

**Induction step.** Suppose that the lemma holds for vertices at height \(h\); we prove that it holds also for any vertex \(v\) at height \(h + 1\).

By the inductive hypothesis and by the algorithm, processes accessing \(v\) from the left son have distinct temporary names in the range \(\{0, \ldots, 2K - 2\}\), and processes accessing \(v\) from the right son have distinct names in the range \(\{2K - 1, \ldots, 4K - 3\}\). The total number \(k\) of processes which access vertex \(v\) does not exceed \(K\), and they have distinct names in the range \(\{0, \ldots, 4K - 3\}\). By Theorem 5.5, the processes obtain distinct names in the range \(\{0, \ldots, 2k - 2\}\), after executing \(\text{net}[v]\).

By this lemma, processes which complete \(\text{net}\) at the root acquire distinct names in \(\{0, \ldots, 2k - 2\}\). Each process performs procedure \(\text{net}\) in \(h = \lceil 2 \log K \rceil\) vertices of the tree. In each vertex a process performs procedure \(\text{net}\) in \(O(K)\) operations (Theorem 5.5). The copies of \(\text{net}\) require only dynamic single-writer single-reader registers, but the preliminary stage of calling \(\text{Adaptive}_k(k+1)/2\)-renaming (Line 1) requires multi-writer multi-reader registers. Thus we have the following theorem:

**Theorem 6.2** Algorithm \(6\) solves \((2k - 1)\)-renaming with fast step complexity \(O(K \log K)\), using multi-writer multi-reader registers.

7 Adaptive \(O(k \log k)\) Algorithm for \((6k - 1)\)-Renaming

In this section, we combine the adaptive lattice agreement algorithm (Algorithm 4) with the fast \((2k - 1)\)-renaming algorithm (Algorithm 6) to obtain an adaptive renaming algorithm with name space of size \(6k - 1\).

Processes first obtain comparable views using adaptive lattice agreement. Then, processes are partitioned into disjoint sets according to the size of their views: \(p_i\) belongs to a set \(S_j\), if
Figure 11: Structure of the fast $(2k - 1)$-renaming algorithm.

**Algorithm 6** Fast $(2k - 1)$-renaming.

Procedure `fastRenaming()`

1. `temp_name := Adaptiveₖ(k + 1)/2-renaming()`
2. `v := 2ʰ + |temp_name/2|`  // ʰ is the height of the tree
3. `side := temp_name mod 2`
4. `temp_name := 0`
5. for(ʰ = ʰ, . . . , 0)
6. \[ temp_name := \text{net}[v](temp_name + side \cdot (2K - 1)) \]
7. `side := v mod 2`
8. `v := |v/2|`
9. `decide(temp_name)`
Lemma 5.4 with \( e = N - 1 \) implies that for each process \( p_i \) leaving the network,

\[-N \leq \text{row}(p_i, N - 1) < -N + 2|S_{N-1}| - 1.\]

Since \(|S_{N-1}| \leq k\), the names chosen in Line 11 are in the range \( \{0, \ldots, 2k - 2\} \).

A process \( p_i \) accesses at most \( 2 \cdot \text{name}_i + 1 \) reflectors in column \( \text{name}_i \), and exactly one reflector in each column \( \text{name}_i + 1, \ldots, N - 1 \). Since each reflector is executed with a constant number of operations, the step complexity of the algorithm is \( O(N) \). Thus we have:

**Theorem 5.5** Algorithm 5 solves one-time \((2k - 1)\)-renaming using \( O(N) \) read and write operations on dynamic single-writer single-reader registers.

The network consists of \( O(N^2) \) reflectors and each reflector is implemented with two dynamic single-writer single-reader registers of size 1 bit. Thus, the algorithm requires \( O(N^2) \) one-bit shared registers.

### 6 Fast \( O(K \log K) \) Algorithm for \((2k - 1)\)-Renaming

The fast \( O(K \log K) \) algorithm for \((2k - 1)\)-renaming works in two stages: In the first stage, processes acquire new names in the range \( \{0 \ldots k(k + 1)/2 - 1\} \), using adaptive \( k(k + 1)/2 \)-renaming (Algorithm 3). In the second stage, processes reduce the size of their name space from \( k(k + 1)/2 \) to \( 2k - 1 \), recursively using the \( O(N) \) algorithm for \((2k - 1)\)-renaming (Algorithm 5). If processes would start the \( O(N) \) algorithm for \((2k - 1)\)-renaming immediately after acquiring new names in \( \{0 \ldots k(k + 1)/2 - 1\} \), then the overall step complexity of the algorithm would be \( O(K^2) \). Instead, we gradually reduce the size of the name space, in \( O(\log K) \) iterations.

The algorithm uses copies of Algorithm 5 arranged in complete binary tree of height \( h = \lceil \log K(K+1)/2 \rceil - 1 \). With each vertex \( v \) of the tree, we associate a distinct copy of Algorithm 5, denoted by \( \text{net}[v] \). This copy is designed for initial name space of size \( N = 4K - 2 \); that is, the network used in \( \text{net}[v] \) has \( 4K - 2 \) columns. Since the algorithm is fast, the maximal number \( K \) of active processes is known in advance.

The vertices of the tree are numbered recursively as follows: The root is numbered 1; if vertex \( v \) is numbered \( \ell \), then its left son is numbered \( 2\ell \), and its right son is numbered \( 2\ell + 1 \).

Process \( p_i \) starts by acquiring a temporary name performing \( O(k^2) \)-renaming. Using this temporary name, \( p_i \) determines the leaf of the tree at which it starts to climb up to the root. At each vertex \( v \) on the path from the leaf to the root, \( p_i \) performs \( \text{net}[v] \). (See Figure 11.) The number of the column in which \( p_i \) starts \( \text{net}[v] \) is determined according to the temporary name obtained at the previous vertex: If \( p_i \) ascends from the left subtree of \( v \), then it starts at one of the first \( 2K - 1 \) columns of the network; otherwise, it starts at one of the last \( 2K - 1 \) columns. The output of the algorithm is the temporary name obtained at the root.

Pseudocode appears in Algorithm 6.
Since $S_{c+1} = S_c \cup \{q\}$, $|S_{c+1}| = |S_c| + 1$. Therefore,
\[
-((c+1)+1) \leq \text{row}(q,c+1) < -((c+1)+1) + 2|S_{c+1}| - 2
\] (1)

According to the algorithm, for each process $p_i \in S_c$,
\[
\text{row}(p_i,c+1) = \begin{cases} 
\text{row}(p_i,c) + 1 & \text{if } \text{exit}(p_i,c+1) = \text{up}_0 \\
\text{row}(p_i,c) - 1 & \text{otherwise}
\end{cases}
\]

Together with the inductive hypothesis, this implies
\[
\text{row}(p_i,c+1) \geq \text{row}(p_i,c) - 1 \\
\geq -(c+1) - 1 = -((c+1)+1)
\] (2)

On the other hand,
\[
\text{row}(p_i,c+1) \leq \text{row}(p_i,c) + 1 \\
< -(c+1) + 2|S_c| - 1 + 1 \\
= -(c+1+1) + 2(|S_{c+1}| - 1) - 1 + 1 \\
= -(c+1)+1+2|S_{c+1}| - 1
\] (3)

The lemma follows from inequalities 1, 2 and 3.
Register \( R_i \) of a reflector is written only by the process entering the reflector on entrance \( in_i \). For any reflector in column \( c \), entrance \( in_i \) is accessed only by the unique process which starts on column \( c \), and entrance \( in_i \) is accessed by at least one process (by Lemma 5.3). Therefore, we can implement the network using only dynamic single-writer single-reader registers.

The next lemma shows that for any column \( c \) in the network, processes in \( S_c \) leave column \( c \) on the lower \( 2|S_c| - 1 \) rows.

**Lemma 5.4** For any \( c = 0, \ldots, N - 1 \), and for any process \( p_i \) in \( S_c \),
\[-(c + 1) \leq \text{row}(p_i, c) < -(c + 1) + 2|S_c| - 1.\]

**Proof:** The proof is by induction on \( c \). In the base, case \( c = 0 \). If there is a process \( p_i \) such that \( \text{name}(p_i) = 0 \), then by Lemma 5.1 \( \text{exit}(p_i, 0) = \text{down}_1 \), since there is no process which accesses reflector \( S[0,0] \) on \( in_0 \). Therefore, by the algorithm we have \( \text{row}(p_i, 0) = -1 \), and the lemma holds.

For the inductive step, suppose that the lemma holds for column \( c \). We consider two cases:

**Case 1:** If no process starts on column \( c + 1 \), then no process accesses reflectors in column \( c + 1 \) on entrance \( in_1 \) (see Figure 10 (a)). Therefore, by Lemma 5.1 each process \( p_i \) in \( S_c \) exits column \( c + 1 \) on \( \text{down}_0 \), and therefore \( \text{row}(p_i, c + 1) = \text{row}(p_i, c) - 1 \); by the inductive hypothesis:
\[
\begin{align*}
\text{row}(p_i, c + 1) &= \text{row}(p_i, c) - 1 \\
&\geq -(c + 1) - 1 \\
&= -((c + 1) + 1)
\end{align*}
\]
On the other hand, since \( |S_{c+1}| = |S_c| \),
\[
\begin{align*}
\text{row}(p_i, c + 1) &= \text{row}(p_i, c) - 1 \\
&< -(c + 1) + 2|S_c| - 1 - 1 \\
&= -((c + 1) + 1) + 2|S_{c+1}| - 1
\end{align*}
\]
The lemma follows from these inequalities.

**Case 2:** Suppose there is a process \( q \) which starts on column \( c + 1 \); that is, \( \text{name}(q) = c + 1 \). By the inductive hypothesis, processes from \( S_c \) access only the lower \( 2|S_c| - 1 \) reflectors in column \( c + 1 \). Since no process accesses the upper reflectors \( S[c + 1, c + 1], S[c + 1, c], \ldots, S[c + 1, -(c + 1)] \) on \( in_0 \), by Lemma 5.1, \( q \) passes these reflectors until it reaches a reflector \( S[c + 1, r'] \) accessed by another process, or until it reaches the lowest reflector \( S[c + 1, -(c + 1)] \) in the column (see Figure 10 (b)). Therefore, \( q \) can leave column \( c + 1 \) either on exit \( \text{down}_1 \) of reflector \( S[c + 1, -(c + 1)] \) or on exit \( \text{up}_0 \) of a reflector \( S[c + 1, r'] \), where \( -(c + 1) \leq r < -(c + 1) + 2|S_c| - 1 \). By the algorithm, this implies
\[
-((c + 1) + 1) \leq \text{row}(q, c + 1) < -(c + 1) + 2|S_c| - 1
\]
Now consider process \( q \). By the algorithm, \( q \) may leave column \( c + 1 \) through exit \( \text{down}_1 \) of the lowest reflector in the column, \( S[c + 1, -(c + 1)] \) (Figure 9(a)), or through exit \( \text{up}_1 \) of a reflector \( S[c + 1, r'] \), where \( -(c + 1) \leq r' \leq c + 1 \) (Figure 9(b)).

If \( q \) leaves reflector \( S[c + 1, -(c + 1)] \) through \( \text{down}_1 \), then by the algorithm,
\[
\text{row}(q, c + 1) = -(c + 1) - 1 = r' - 1.
\]

If there is a process \( p_j \in S_c \) which also accesses \( S[c + 1, -(c + 1)] \), then by Lemma 5.2, \( p_j \) leaves \( S[c + 1, -(c + 1)] \) through exit \( \text{up}_0 \) and therefore,
\[
\text{row}(p_j, c + 1) = -(c + 1) + 1 = r' + 1.
\]

If \( q \) leaves reflector \( S[c + 1, -(c + 1)] \) through \( \text{up}_1 \), then by the algorithm, \( \text{row}(q, c + 1) = r' \). By Lemma 5.2, there is a process \( p_j \in S_c \) which executes \( S[c + 1, r'] \); that is, \( \text{row}(p_j, c) = r' \). By the algorithm
\[
\text{row}(p_j, c + 1) = \begin{cases} 
  r' - 1 & \text{if } p_j \text{ leaves through } \text{down}_0 \\
  r' + 1 & \text{otherwise}
\end{cases}
\]

The inductive hypothesis and the above equations imply that for any pair of processes \( p_i, p_j \in S_c \), \( \text{row}(p_i, c + 1) \neq \text{row}(p_j, c + 1) \).

Lemma 5.3 with \( c = N - 1 \) implies that the names acquired by the processes are distinct.
Lemma 5.2 If a single process enters the 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 \text{true} from \( R_1 \) and \( R_0 \), then by the algorithm, \( \text{exit}(p_i) = \text{up}_0, \text{exit}(p_j) = \text{up}_1 \), and the lemma holds.

Otherwise, without loss of generality, \( p_i \) reads \text{false} from \( R_1 \). Since \( p_i \) reads \text{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 \text{true} from \( R_0 \) and by the algorithm, \( \text{exit}(p_j) = \text{up}_1 \), which proves the lemma.

For any \( c = 0, \ldots, N - 1 \), let \( S_c \) be the set of active processes which start at columns \( 0, \ldots, c \). For any process \( p_i \in S_c \), let \( \text{row}(p_i, c) \) be the value of the local variable \text{row} before \( p_i \) accesses the first reflector in column \( c + 1 \).

Lemma 5.3 For a column \( c = 0, 1, \ldots, N - 1 \), and for any pair of processes \( p_i \) and \( p_j \) in \( S_c \), \( \text{row}(p_i, c) \neq \text{row}(p_j, c) \).

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 is true for column \( c \); we will show that the lemma holds also for column \( c + 1 \).

If there is no process which starts on column \( c + 1 \), then by the algorithm no reflector in column \( c + 1 \) is accessed on entrance \( in_1 \). By Lemma 5.1, any process \( p_i \in S_c \) leaves column \( c + 1 \) through exit \( \text{down}_0 \). By the algorithm, we have

\[
\text{row}(p_i, c + 1) = \text{row}(p_i, c) - 1
\]

and the lemma follows by the inductive hypothesis.

Now suppose that there is a process \( q \) such that \text{name}(q) = c + 1; that is, \( q \) starts on column \( c + 1 \). Let \( S[c + 1, r'] \) be the last reflector accessed by \( q \) in column \( c + 1 \). This means that \( q \) leaves reflectors \( S[c + 1, c + 1], S[c + 1, c], \ldots, S[c + 1, r' + 1] \) of column \( c + 1 \) through exit \( \text{down}_1 \), and \( q \) does not access any of reflectors \( S[c + 1, r' - 1], S[c + 1, r' - 2], \ldots, S[c + 1, -(c + 1)] \).

By Lemma 5.2, any process \( p_i \) in \( S_c \) which enters column \( c + 1 \) on row \( r' + 1 \) or higher, exits the column on \( \text{up}_0 \). Therefore by the algorithm,

\[
\text{row}(p_i, c + 1) = \text{row}(p_i, c) + 1 > r' + 1.
\]

By Lemma 5.1, any process \( p_i \) in \( S_c \) which enters column \( c + 1 \) on row \( r' - 1 \) or lower, exits the column on \( \text{down}_0 \). Consequently,

\[
\text{row}(p_i, c + 1) = \text{row}(p_i, c) - 1 < r' - 1.
\]
Algorithm 5 $(2k-1)$-renaming.

Procedure net($name_i : 0, \ldots, N-1$) \hspace{1cm} // renaming algorithm
Initially, $in := 1$; $col := name_i$; $row := name_i$
private variables:
    $in$: integer, initially 1;
    $col$: integer, initially $name_i$;
    $row$: integer, initially $name_i$;
1. while($in = 1$) \hspace{1cm} // descend through column $name_i$
2. $exit := \text{reflector}[row, col](in)$
3. if ($exit = up_1$) then
4.    $col++$; $in:= 0$;
5. else
6.    $row--;$ \hspace{1cm} // $exit = down_1$
7. if ($row < -col$) then
8.    $col++$; $in:=0$; \hspace{1cm} // the lowest reflector is reached
9. while ($col < N$) \hspace{1cm} // move towards column $N-1$
10. $exit := \text{reflector}[row, col](in)$
11. if ($exit = up_0$) then $col++$; $row++$;
12. else $col++$; $row--;$ \hspace{1cm} // $exit = down_0$
13. decide($row + N$);

Procedure reflector(entrance $i : 0, 1$)
1. write($R_i, \text{true}$);
2. if ($R_{i-1} = \text{false}$) then $exit := down_i$
3. else $exit := up_i$
4. return($exit$)
Figure 8: The network of reflectors for \((2k - 1)\)-renaming.
5 An $O(N)$ Algorithm for $(2k - 1)$-Renaming

The algorithm is based on a network of simple building blocks called reflectors. A reflector is very similar to splitter: a process entering the reflector changes the direction of its movement according to the information obtained. The difference is that reflector has two distinguished entrances and the direction of the further movement depends also on the entrance through which the process accesses the reflector.

The entrances of a reflector are denoted by $i_{n_0}$ and $i_{n_1}$. The reflector also has two lower exits, $down_0$ and $down_1$, and two upper exits, $up_0$ and $up_1$. A process entering the reflector on entrance $i_{n_0}$ leaves the reflector only through exits $up_0$ or $down_1$ (see top left corner of Figure 8).

A simple implementation of a reflector associates two one-bit registers $R_0$ and $R_1$ with entrances $i_{n_0}$ and $i_{n_1}$, respectively. Initially $R_0 = R_1 = false$. Process $p_i$ entering the reflector on entrance $i_{n_0}$, writes true into register $R_i$ and then reads the other register, $R_{1-i}$; if it is false, $p_i$ takes the lower exit; otherwise, $p_i$ takes the upper exit. Clearly, the complexity of the implementation is $O(1)$.

The algorithm uses a network with $N$ columns; each column $c$ contains $2c - 1$ reflectors (see Figure 8). Process $p_i$ with initial name name$_c$ starts by entering the upper reflector of column name$_c$ on entrance $i_{n_1}$. The process descends through the reflectors of the column until it sees another process in a reflector, or it reaches the lowest reflector of the column. After that, the process moves towards column $N - 1$. In each column, except its first column, a process executes exactly one reflector, entering it on entrance $i_{n_0}$.

Specifically, the lower exit $down_1$ of a reflector $S[c, r]$ is connected to entrance $i_{n_1}$ of reflector $S[c, r - 1]$, if $r > c$, and it is connected to entrance $i_{n_0}$ of reflector $S[c + 1, r - 1]$, if $r = c$ (that is, $S[r, c]$ is the lowest reflector of column $c$). The upper exit $up_1$ of $S[c, r]$ is connected to entrance $i_{n_0}$ of $S[c + 1, r]$. The lower exit $down_0$ of $S[c, r]$ is connected to entrance $i_{n_0}$ of $S[c + 1, r - 1]$. The upper exit $up_0$ of $S[c, r]$ is connected to entrance $i_{n_0}$ of $S[c + 1, r + 1]$, for any $0 \leq c < n - 1$ and $-c \leq r \leq c$.

As we shall see, each process leaves the network on a different row, and the row numbers provide distinct names in a range of size $2k - 1$.

Pseudocode appears in Algorithm 5.

Suppose that $p_i$ enters the reflector on entrance $i_{n_1}$, and no process enters the reflector on entrance $i_{n_0}$. Since no process writes to $R_{1-i}$, $p_i$ reads false from $R_{1-i}$ and leaves the reflector through the lower exit, down$_i$. Thus, we have the following lemma:

**Lemma 5.1** If a single process accesses a reflector, then it leaves through a lower exit.

Similar type of arguments are used in the following lemma:

---

$^2$Note, that a splitter is a reflector in which $i_{n_0}$ is identified with $i_{n_1}$, $down_0$ is identified with $down_1$, and $up_0$ is identified with $up_1$. 

---
Lemma 4.3 If there are \( k \) active processes, then processes execute \( \text{RestrictedLA}_{2^j} \) in Line 3 of Algorithm 4 only for \( j \leq \lceil 2 \log k \rceil \).

For any \( j \), \( 0 \leq j < \lceil 2 \log k \rceil \), \( \text{RestrictedLA}_{2^j} \) is accessed only by processes which obtained temporary names \( 2^i \leq \text{temp}_\text{name} < 2^{i+1} \) in \( \text{Adaptive}_k(k+1)/2\text{-renaming} \) (Line 1). Since all the temporary names acquired in the renaming algorithm are distinct, \( \text{RestrictedLA}_{2^j} \) is accessed by at most \( 2^{i+1} - 2^i = 2^i \) processes with distinct names in the range \( \{0\ldots2^i-1\} \).

By Lemma 4.2, we have:

Lemma 4.4 For any \( j \), \( 0 \leq j < \lceil 2 \log k \rceil \), the views of processes exiting \( \text{RestrictedLA}_{2^j} \) satisfy the properties of the lattice agreement problem.

We now show that the lattice agreement properties hold after executing \( \text{union}_j \) in Lines 4 and 6.

Lemma 4.5 For any \( j \), \( 0 \leq j \leq \lceil 2 \log k \rceil \), the following holds:

1. The views of processes exiting \( \text{union}_j \) are comparable;
2. The output view of a process exiting \( \text{union}_j \) includes its input view;

Proof: We prove the lemma by backward induction on \( j \).

Base case: \( j = \lceil 2 \log k \rceil \). For any \( l > \lceil 2 \log k \rceil \), no process executes \( \text{RestrictedLA}_{2^l} \), by Lemma 4.3. Therefore, no process performs procedures \( \text{union}_l \), \( l > \lceil 2 \log k \rceil \). It follows that there is no process which accesses \( \text{union}_j \) on entrance \( \text{in}_1 \). By Lemma 4.4, the views of processes entering \( \text{union}_j \) on entrance \( \text{in}_0 \) are comparable. Therefore, Lemma 4.1 implies that statements (1) and (2) hold.

Induction step: Suppose that the lemma holds for some \( j > 0 \). Then, by the inductive hypothesis the views of the processes entering \( \text{union}_{j-1} \) on entrance \( \text{in}_1 \) are comparable. The views of the processes entering \( \text{union}_{j-1} \) on entrance \( \text{in}_0 \) are comparable by Lemma 4.4. Therefore, by Lemma 4.1, conditions (1) and (2) hold.

Lemma 4.5 with \( j = 0 \) implies that the properties of lattice agreement hold when processes exit \( \text{union}_0 \).

We now compute the total step complexity of the algorithm. In Line 1 a process performs \( \text{Adaptive}_k(k+1)/2\text{-renaming} \) in \( O(k) \) operations (Theorem 3.12). Then, in Line 3 the process executes \( \text{RestrictedLA}_{2^j} \), for \( j \leq \lceil 2 \log k \rceil \) (Lemma 4.3), and this requires \( O(kj) = O(k \log k) \) operations. In addition, in Lines 4–6 the process performs at most \( \lceil 2 \log k \rceil \) copies of \( \text{union}_j \) (Lemma 4.3), and each copy requires \( O(k) \) operations (Lemma A.2). Thus, the total step complexity of the algorithm is \( O(k \log k) \). Therefore, we have the following theorem:

Theorem 4.6 Algorithm 4 solves lattice agreement with adaptive step complexity \( O(k \log k) \).
Denote by $OutViews(v)$ the set of views returned by $\text{union}[v]$. Denote by $V_{in,v}(p)$ and $V_{out,v}(p)$ the local views of a process $p$ before and after execution of $\text{union}[v]$. Denote by $V_{in}(p)$ the input parameter of $p$ before execution of $\text{RestrictedLA}$. We say that two views are comparable if the sets of id’s represented by these views satisfy the inclusion relation.

The following lemma shows that for any vertex $v$, the views obtained by the processes that execute $\text{union}[v]$ at Line 12 satisfy the lattice agreement properties. The proof is similar to the proof of Lemma 6 from [15].

**Lemma 4.2** If the processes executing $\text{RestrictedLA}_j$ have distinct names in the range $\{0 \ldots 2^j - 1\}$, then the following conditions hold for any vertex $v$:

1. the views of the processes exiting $\text{union}[v]$ are comparable, and
2. the output of a process exiting $\text{union}[v]$ contains its input view.

**Proof:** The proof of the lemma is by induction on height $h$ of $v$.

Let $InViews_0(v)$ and $InViews_1(v)$ be the sets of the views of the processes that access $\text{union}[v]$ on entrances $in_0$ and $in_1$, respectively.

**Base case:** $h = 0$. By the algorithm, if $v$ is a leaf, then at most one process may access $\text{union}[v]$ on each of its entrances, since the processes have distinct names in $\{0, \ldots, 2^j - 1\}$. Therefore, the inclusion relation trivially holds for the views in $InViews_0(v)$ and in $InViews_1(v)$. Conditions (1) and (2) follow from Lemma 4.1.

**Induction step.** Suppose that the lemma holds for any vertex at height $h$. We prove that it also holds for a vertex $v$ at height $h + 1$.

Let $u$ be the left child of $v$ and let $w$ be the right child of $v$. By the algorithm, $InViews_0(v) = OutViews(u)$ and $InViews_1(v) = OutViews(w)$. By the inductive hypothesis, any pair of views in $OutViews(u)$ and in $OutViews(w)$ are comparable. Therefore by Lemma 4.1 the views in $OutViews(v)$ are comparable.

By the algorithm, for any process $p$ which accesses $\text{union}[v]$ on entrance $in_0$, $V_{out,w}(p) = V_{in,v}(p)$. For any process $p$ which accesses $\text{union}[v]$ on entrance $in_1$, $V_{out,w}(p) = V_{in,v}(p)$. Therefore, $V_{in}(p) \subseteq V_{out,w}(p)$, by the inductive hypothesis and Lemma 4.1.

By Lemma 4.2, processes that complete $\text{union}$ at the root obtain views that satisfy the lattice agreement properties. Each process accesses $j$ vertices on the path from a leaf to the root. Execution of $\text{union}$ at each vertex requires $O(k)$ operations (Lemma A.2). Therefore, the complexity of $\text{RestrictedLA}_j$ is $O(jk)$.

We now prove that $\text{AdaptiveLA}$ presented in Algorithm 4 solves the lattice agreement problem with adaptive step complexity $O(k \log k)$.

Suppose that there are $k$ active processes. After performing adaptive $k(k + 1)/2$-renaming in Line 1, a process $p_k$ gets a unique $\text{temp\_name}_k$ in the range $\{1 \ldots k(k + 1)/2 - 1\}$. In Line 3 the process performs the $[\log(\text{temp\_name})]$-th copy of $\text{RestrictedLA}$, in Line 3. This implies the following lemma:
Algorithm 4 Adaptive lattice agreement.

Procedure AdaptiveLA() // entry point of the algorithm
1. \( temp\_name := Adaptive \_k(k + 1)/2\text{-renaming}() + 1 \)
2. \( r := \lfloor \log(temp\_name) \rfloor \)
3. \( V := RestrictedLA_2(temp\_name - 2^r) \)
4. \( V := \text{union}[v](V, 0) \)
5. for \( j = r - 1 \) to 0 do
6. \( V := \text{union}_j(V, 1) \)
7. decide(\( V \))

Procedure RestrictedLA_2(name_i : 0, \ldots, 2^j - 1)
8. \( v := 2^h + \lfloor name_i/2 \rfloor \)
9. \( side := name_i \mod 2 \)
10. \( V := \{p_i\} \)
11. for \( (l = h, \ldots, 0) \)
12. \( V := \text{union}[v](V, side) \)
13. \( side := v \mod 2 \)
14. \( v := \lfloor v/2 \rfloor \)
15. decide(\( V \))

Figure 7: Structure of the adaptive lattice agreement algorithm.
 RestrictedLA_{j} uses a complete binary tree of height \( h = j - 1 \) (instead of a tree of height \( \lfloor \log n \rfloor - 1 \), used in [15]). A distinct copy of the adaptive unifying procedure is associated with each vertex \( v \) of the tree, and it is denoted \( \text{union}[v] \).

The vertices of the tree are numbered recursively as follows: The root is numbered 1. If vertex \( v \) is numbered \( l \), then the left child of \( v \) is numbered \( 2l \), and the right child of \( v \) is numbered \( 2l + 1 \) (see Figure 6).

Process \( p_i \) with initial name \( x \) starts \( \text{RestrictedLA}_{j} \) at the \( \lfloor x/2 \rfloor \)-th leaf of the tree and climbs up to the root. At each level, \( p_i \) performs \( \text{union} \) using the view obtained in the previous level as input parameter. The output of \( \text{RestrictedLA}_{j} \) is the view obtained at the root.

Finally, the adaptive lattice agreement algorithm is built from the following components:

- the adaptive \( k(k+1)/2 \)-renaming;
- \( \lfloor \log n(n+1)/2 \rfloor \) copies of \( \text{union} \), denoted \( \text{union}_0, \text{union}_1, \ldots \);
- \( \lfloor \log n(n+1)/2 \rfloor \) copies of \( \text{RestrictedLA} \), denoted \( \text{RestrictedLA}_0, \text{RestrictedLA}_1, \ldots \).

A process \( p_i \) starts by acquiring a new name using the adaptive \( k(k+1)/2 \)-renaming algorithm, and determines in which copy of the lattice agreement algorithm it should participate. By executing the appropriate copy \( \text{RestrictedLA}_{j} \), \( p_i \) obtains a view \( V(p_i) \). Then \( p_i \) enters \( \text{union} \) on entrance \( in_0 \), using \( V(p_i) \) as input parameter. The exit of \( \text{union}_j, j > 0 \), is connected to entrance \( in_1 \) of \( \text{union}_{j-1} \); \( p_i \) performs \( \text{union}_{j-1}, \ldots, \text{union}_0 \), entering them on entrance \( in_1 \), and decides on the view obtained in \( \text{union}_0 \).

Figure 7 presents the overall scheme of the algorithm. Pseudocode appears in Algorithm 4.

We start by proving the correctness of \( \text{RestrictedLA}_{j} \).
If \( k \) processes execute the algorithm, then by Lemma 3.10, all processes stop at most \( k - 1 \) steps away from \((0,0)\). Consequently, by Lemma 3.11, each process obtains a distinct name in the range \( \{0, \ldots, k(k+1)/2 - 1\} \). The implementation of a splitter, the algorithm which controls the movement of processes in the grid and the numbering of the splitters do not relay on knowledge of \( k \). Therefore, we have the following theorem:

**Theorem 3.12** Algorithm 3 solves \( k(k+1) \)-renaming with adaptive step complexity \( O(k) \).

### 4 Adaptive \( O(k \log k) \) Lattice Agreement

In the previous section we described a simple algorithm for lattice agreement with adaptive step complexity \( O(k^2) \). In this section we present a lattice agreement algorithm with step complexity \( O(k \log k) \).

The adaptive lattice agreement algorithm works as follows. First, the processes reduce the name space to a size which depends only on \( k \) (using an adaptive \( k(k+1)/2 \)-renaming algorithm). After that, each process estimates the number of participants based on its new name, rounded to the closest integral power of 2; then the process performs a copy of a fast lattice agreement algorithm, designed for the same number of participants (the copy is denoted RestrictedLA\(_j\), if it is designed for \( 2^j \) active processes). Finally, the views obtained in each copy of RestrictedLA are merged using a union procedure.

An adaptive \( k(k+1)/2 \)-renaming algorithm with step complexity \( O(k) \) was presented in Subsection 3.3 (Algorithm 3).

Procedure union was proposed by Inoue, Chen, Masuzawa and Tokura [15]. The procedure takes two parameters, a view \( V \) and an integer \( \text{side} \in \{0,1\} \), and returns a view. Let \( S_i \) be the set of processes which call union with parameter \( \text{side} = i \); we say also that they enter union on entrance \( in_i \). The following lemma, from [15], states the main property of union:

**Lemma 4.1 (Inoue, Chen, Masuzawa and Tokura)** If the input parameters of the processes in \( S_i \), \( i = 0,1 \), are comparable, then the views returned by union are also comparable.

The implementation of union is described in Appendix A, where we also argue that it has adaptive step complexity \( O(k) \) (Lemma A.2).

RestrictedLA\(_j\), is a modification of the linear lattice agreement algorithm of Inoue, Chen, Masuzawa and Tokura [15]. In [15], it is assumed that \( n \) processes with names in the range \( \{0 \ldots n - 1\} \) execute the algorithm. We adapt this algorithm so that RestrictedLA\(_j\) works correctly when executed by \( k \) processes with names in range \( \{0 \ldots 2^j - 1\} \). (Note, that different copies of RestrictedLA are used for initial name spaces of different size.) The complexity of RestrictedLA\(_j\), is \( O(jk) \); \( k \), the number of active processes, need not be known in advance.
Algorithm 3 Adaptive \( k(k+1)/2 \)-renaming: code for process \( p_i \)

Procedure Adaptive \( k(k+1)/2 \)-renaming()

private variables:
- \( i \) integer initially 0
- \( j \) integer initially 0
- \( move: \) \{left, right, stop\} initially down

1. while \( move \neq stop \)
2. \( move := \text{Splitter}[i,j]() \)
3. if \( move = \text{down} \) then \( i++ \)
4. else if \( move = \text{right} \) then \( j++ \)
5. return \((i + j)(i + j + 1)/2 + j)\)

Let \((i, j)\) be grid position at the \(i\)th row and \(j\)th column, \(0 \leq i < n\) and \(0 \leq j < n\). The top left corner of the grid is \((0, 0)\).

A splitter in grid position \((i, j)\) is associated with the name \((i + j)(i + j + 1)/2 + j\), as shown in Figure 5.

An implementation of the grid is presented in Algorithm 3.

We say that grid position \((i, j)\) is \(i + j\) steps away from grid position \((0, 0)\). The following lemma, from [17], states the main property of the grid.

Lemma 3.10 If a set of \( k \) processes access the grid, then each process stops after \( O(k) \) operations in a unique grid position which is at most \( k - 1 \) steps away from \((0, 0)\).

The following lemma shows that the numbering of the splitters assigns names which depend only on the distance of the splitters from grid position \((0, 0)\).

Lemma 3.11 Splitters that are at most \( l - 1 \) steps away from grid position \((0, 0)\) have names in the range \( \{0, \ldots, l(l+1)/2-1\} \), for any \(0 < l \leq n\).

Proof: For every \( h, 0 \leq h < n\), all the splitters that are \( h \) steps away from \((0, 0)\) belong to the same diagonal in the grid, denoted \( d_h \) (see Figure 5). According to the numbering of the splitters, the names associated with the splitters on diagonal \( d_h \) are greater than the names associated with the splitters on diagonal \( d_{h-1} \), and less than the names associated with the splitters on diagonal \( d_{h+1} \).

Diagonal \( d_h \) contains exactly \( h + 1 \) splitters. Therefore, diagonals \( d_0, \ldots, d_{n-1} \) contain a total of \( \sum_{h=0}^{n-1} (h + 1) = l(l+1)/2 \) splitters, numbered from 0 to \( l(l+1)/2 - 1\). 

16
Step complexity. The maximal size of the view returned by collect is $k$. A process calls collect only if its local view increased in the previous iteration. Since the maximal size of the view returned by collect is $k$, a process calls collect at most $k + 1$ times. The step complexity of collect is $O(k)$, and therefore, the step complexity of Algorithm 2 is $O(k^2)$. ■

3.3 Adaptive $O(k^2)$-Renaming in $O(k)$ Operations

In this subsection we show another application of the splitter suggested by Moir and Anderson [17] — an adaptive $k(k + 1)/2$-renaming algorithm. In this algorithm the splitters are arranged in a square grid as in [17] (see Figure 5).

Each process starts the algorithm by executing the splitter in the upper left corner of the grid. According to the result obtained in a splitter, a process moves right or down in the grid and continues execution, or stops in the current splitter and acquires the name associated with the splitter. The grid spreads the processes so that finally each process stops in a distinct splitter.

Since as many as $n$ processes may participate in the algorithm, we use a square grid of size $n \times n$. The splitters are numbered by diagonals, while Moir and Anderson number the splitters by rows, from left to right.
Algorithm 2 Adaptive lattice agreement: code for process \( p_i \)

Procedure \text{LatticeAgreement}(\) 
\begin{itemize} 
\item registration() 
\item \( V_i := - \) 
\item \( V_i := \text{collect}() \) 
\item while \( (V_i \neq V_i') \) 
\item \( V_i := V_i' \) 
\item \( V_i := \text{collect}() \) 
\item return \( (V_i) \) 
\end{itemize} 

Figure 4: Illustration for the proof of Lemma 3.9

The code is shown in Algorithm 2.

**Theorem 3.9** Algorithm 2 solves the lattice agreement problem with adaptive step complexity \( O(k^2) \).

**Proof:** *Comparability.* By Property (2) of \text{collect} (Lemma 3.5), its output contains only identifiers of active processes. Therefore, \( V_i \) is a view.

Let \( V_{out}(p_i) \) and \( V_{out}(p_j) \) be the views obtained by processes \( p_i \) and \( p_j \) in Algorithm 2. Let \( cl_i \) and \( cl'_i \) be the last two executions of \text{collect} performed by \( p_i \); let \( cl_j \) and \( cl'_j \) be the last two executions of \text{collect} performed by \( p_j \).

Note that by the algorithm, \( cl_i \) and \( cl'_i \) return the same view \( V_{out}(p_i) \), and \( cl_j \) and \( cl'_j \) return the same view \( V_{out}(p_j) \).

By the algorithm, \( p_k \) completes \( cl_i \) before it starts \( cl'_i \); \( p_j \) completes \( cl_j \) before it starts \( cl'_j \). This implies that either \( cl_i \) finishes before \( cl'_j \) starts, or \( cl_j \) finishes before \( cl'_i \) starts (see Figure 4). In either case, the returned views are comparable by Property (3) of Lemma 3.5.

*Self containment.* Each process \( p_k \) returns the view obtained in the last \text{collect}. Since \( p_k \) completes \text{registration} before it starts the last \text{collect}, by property (1) of Lemma 3.5, \( p_k \) returns a view which satisfies the self-containment property of lattice agreement.
construction of \( S \), \( x \) appears between \( v_i \) and \( v_{i+1} \) in \( S \), which contradicts the assumption that \( v_{i+1} \) appears immediately after \( v_i \) in \( S \).

Assume, \( x = v_i \). This implies that \( v_{i+1} \) is a descendant of \( v_i \) in the tree. Since \( v_i \) precedes \( v_{i+1} \) in the in-order sequence \( S \), \( v_{i+1} \) belongs to the right subtree of \( v_i \). Since \( v_{i+1} \) appears immediately after \( v_i \) in the sequence, the left child of \( v_{i+1} \) must be unmarked (see Figure 3(b)). Therefore, no process obtains \text{left} in vertex \( v_{i+1} \). Since \( v_{i+1} \) is \text{grey}, no process obtains \text{stop} in the vertex. This implies that all the processes accessing \( v_{i+1} \) obtain \text{right}, which contradicts Property (3) of Lemma 3.1.

The proof of the case \( x = v_{i+1} \) is symmetric.

By Lemma 3.3, every active process writes its id in a single vertex. Therefore, the number of the black vertices is at most \( k \). Then Lemma 3.7 implies that the number of the grey vertices is at most \( k + 1 \). Thus the marked tree contains at most \( 2k + 1 \) vertices.\(^1\)

Therefore, a process performing a DFS traversal of a marked part of the tree visits \( O(k) \) vertices. In each vertex, \( p_i \) executes a constant number of operations. Therefore, we have:

**Lemma 3.8** Procedure \text{collect} has adaptive complexity \( O(k) \).

### 3.2 Adaptive Lattice Agreement in \( O(k^2) \) Operations

The \text{collect} procedure described in the previous subsection can be used to implement a simple adaptive algorithm for the lattice agreement problem.

A process starts by performing \text{registration}; then it repeatedly executes \text{collect} until it gets the same view twice, and decides on the last obtained view. This is a traditional snapshot

\(^1\)By arguments similar to those used in Lemma 3.7, we can show that the first and the last vertices in sequence \( S \) are grey. This implies that the number of grey vertices is at most \( k - 1 \), and hence the total number of marked vertices is at most \( 2k - 1 \).
(2) if a process \( p_j \) starts registration after \( p_i \) completes collect, then \( \text{id}_j \not\in V_i \).

**Proof:**  (1) If \( p_j \) completes registration before \( p_i \) starts collect. Then \( p_j \) writes \( \text{id}_j \) into register \( R[v] \) of some vertex \( v \) before \( p_i \) starts collect. By Lemma 3.4, all the vertices on the path from the root to vertex \( v \) are marked. Therefore, \( p_i \) visits vertex \( v \) during the DFS traversal. By the algorithm, this implies that \( \text{id}_j \in V_i \).

(2) If \( p_j \) starts operation registration after \( p_i \) completes operation collect, then \( p_j \) writes its id into the shared memory after \( p_i \) completes reading the registers. Therefore, \( \text{id}_j \not\in V_i \). ■

The following lemma shows that non-overlapping collect operations return comparable views; the views increase according to the order of the operations.

**Lemma 3.6** Let \( V_i \) be the output of procedure collect performed by a process \( p_i \), and \( V_j \) be the output of procedure collect performed by a process \( p_j \). If \( p_i \) completes the collect before \( p_j \) starts, then \( V_i \subseteq V_j \).

**Proof:** Suppose that \( \text{id}_i \in V_i \). Then process \( p_i \) writes \( \text{id}_i \) into register \( R[v] \) in a vertex \( v \) before \( p_i \) completes operation collect, according to Property (2) of Lemma 3.5. By Lemma 3.3, no other process writes its id to \( v \). Therefore, \( R[v] \) contains \( \text{id}_i \) when \( p_j \) starts its collect. This implies that \( \text{id}_i \in V_j \). ■

Lemma 3.5 and Lemma 3.6 imply that the algorithm solves the collect problem.

We now compute the step complexity of the DFS traversal of the marked tree performed in collect. In general, the size of a tree with depth \( k \) may be exponential in \( k \), but, as we prove below, the marked tree contains at most \( 2^k - 1 \) vertices.

We say that a marked vertex \( v \) is **black**, if the register \( R[v] \) associated with the vertex contains the id of some process \( p_k \) (that is, \( p_k \) stops in \( v \)); a marked vertex which is not black is **grey**.

Let \( S = v_0, v_1, \ldots, v_i \) be the sequence of the vertices obtained by an in-order traversal of the marked tree. By the definition of the in-order traversal, for any vertex \( v \), the vertices that form the left sub-tree of \( v \) appear before \( v \) in \( S \), while the vertices that form the right sub-tree of \( v \) appear after \( v \) in \( S \).

**Lemma 3.7** There is a black vertex between any two grey vertices in the \( S \).

**Proof:** Suppose by contradiction that there are two consecutive grey vertices \( v_i \) and \( v_{i+1} \) in sequence \( S \). Let \( x \) be the lowest common ancestor of \( v_i \) and \( v_{i+1} \) in tree \( T_m \).

We argue that either \( x = v_i \) or \( x = v_{i+1} \). Otherwise, if \( x \neq v_i \) and \( x \neq v_{i+1} \), then \( v_i \) belongs to the left subtree of \( x \), and \( v_{i+1} \) belongs to the right subtree of \( x \) (see Figure 3(a)). By the
Lemma 3.2: If a vertex $v$ has depth $d \geq 0$ in the tree, then at most $k - d$ processes may access $v$ during an execution of the algorithm.

Proof: The proof is by induction on the depth, $d$.

Base case: $d = 0$. The lemma trivially holds, since only $k$ processes participate in the execution.

Induction step. Suppose that the lemma holds for any vertex with depth $d$, $0 \leq d < k$. We will show that it holds also for any vertex $v$ with depth $d + 1$.

Let $u$ be the parent of $v$ in the tree. Suppose that $v$ is the left child of $u$. Since vertex $u$ has depth $d$, by the inductive hypothesis at most $k - d$ processes access $u$. By Property (2) of Lemma 3.1, at most $k - d - 1$ of the processes obtain left at $u$ and access $v$.

The proof is symmetric if $v$ is the right child of $u$.

By Lemma 3.2 and by the algorithm, each process stops in a vertex with a depth less than or equal to $k - 1$. By Property (1) of the splitter (Lemma 3.1), at most one process stops in each vertex. Therefore, we have the following lemma:

Lemma 3.3: If $k$ processes perform registration, then each process writes its id in some vertex with depth less than or equal to $k - 1$ and no other process writes its id in the same vertex.

Since execution of the splitter takes a constant number of operations, Lemma 3.3 implies that the step complexity of registration is $O(k)$.

Let $v$ be a marked vertex. Then some process $p$ sets $mark[v] = 1$. By the algorithm, $p$ accesses vertex $v$ only after it accesses all the vertices on the path from the root to $v$ and sets their $mark$'s to 1. By the algorithm, no process resets $mark[v']$ to 0. Therefore, we have the following lemma:

Lemma 3.4: If some vertex $v$ in the tree is marked, then all the vertices on the path from the root to $v$ are marked.

It follows from Lemma 3.3 and Lemma 3.4 that the marked vertices form a tree whose depth is at most $k$. Therefore, collect is well-defined and it returns all the id's which were written in the vertices before the collect started.

Lemma 3.5: Let $V_i$ be the output of procedure collect performed by a process $p_i$. Then the following conditions hold:

1. if a process $p_j$ completes registration before $p_i$ starts collect, then $id_j \in V_i$;
Algorithm 1 Procedures registration and collect: code for process $p_i$

Procedure registration():
// declare that the process
// participates in the computation

private variables:
- $move$: $\emptyset \cup \{\text{left, right, stop}\}$ initially $\emptyset$
- $v$: vertex initially root

1. while ($move \neq \text{stop}$) do
2. \hspace{1em} $mark[v] := \text{true}$
3. \hspace{1em} $move := \text{Splitter}_v()$ \hspace{1em} // returns stop, left, or right
4. \hspace{2em} if ($move = \text{left}$) then $v := \text{left}_\text{son}(v)$
5. \hspace{2em} if ($move = \text{right}$) then $v := \text{right}_\text{son}(v)$
6. \hspace{1em} $R[v] := \text{id}_i$ \hspace{1em} // now $move = \text{stop}$ — registration in the node $v$

Procedure Splitter(v): \hspace{1em} // from Moir and Anderson [17]

shared variables:
- $X_v$: $\emptyset \cup \{0, \ldots, N\}$ initially $\emptyset$
- $Y_v$: boolean initially false

private variable:
1. $X_v := \text{id}_i$
2. if $Y_v$ then return(right)
3. else $Y_v := \text{true}$
4. \hspace{1em} if $X_v = \text{id}_i$ then return(stop)
5. \hspace{1em} else return(left)

Procedure collect(): \hspace{1em} // collect a subset of active processes
1. $V_i = \text{DFS}(\emptyset, \text{root})$
2. return($V_i$)

Procedure DFS($V$: set of id's; $v$: vertex): \hspace{1em} // DFS traversal of the marked subtree
3. if ($mark[v] \neq 0$) then
4. \hspace{1em} if ($R[v] \neq \emptyset$) then $V := V \cup \text{read}(R[v])$
5. \hspace{2em} $V := V \cup \text{DFS}(V, \text{left}_\text{child}(v))$
6. \hspace{2em} $V := V \cup \text{DFS}(V, \text{right}_\text{child}(v))$
7. return($V$)
Figure 2: An execution of registration in a complete binary tree of splitters

according to the values obtained in the splitters along the path. When the process obtains stop at vertex $v$, it writes its id into the register $R[v]$ associated with $v$ and finishes the procedure (see Figure 2).

A boolean flag $\text{mark}[v]$ is associated with each vertex $v$. Initially, $\text{mark}[v] = 0$ for each vertex $v$. When a process access vertex $v$, it sets $\text{mark}[v] = 1$. A vertex $v$ is marked, if $\text{mark}[v] = 1$.

To perform a collect, process $p_i$ traverses the tree in DFS order, starting at the root of the tree and visiting only marked vertices. During the traversal, $p_i$ collects a set $V_i$ of id’s written in the vertices.

The code appears in Algorithm 1.

The following lemma, from [17], states the main properties of the splitter.

**Lemma 3.1** If $k$ processes execute the splitter, then the following conditions hold:

1. at most one process obtains stop in the splitter;
2. at most $(k-1)$ processes obtain left in the splitter;
3. at most $(k-1)$ processes obtain right in the splitter.
3 Basic Adaptive Algorithms: Collect, Lattice Agreement and $O(k^2)$-Renaming

A solution for the collect problem should define algorithms for registration and collect. In this section we present adaptive $O(k)$ implementations for these operations.

Then, using adaptive registration and collect we build an adaptive algorithm for the lattice agreement problem with step complexity $O(k^3)$.

In this section we also present an adaptive $O(k)$ algorithm for $O(k^2)$-renaming which uses the same basic components as the implementation of registration and collect.

3.1 Adaptive Collect in $O(k)$ Operations

The basic component of the collect algorithm is the building block suggested by Moir and Anderson [17]. To distinguish this building block from other components used in the work, we call it a splitter. A process which executes the code of the splitter, obtains one of the following values: left, right or stop (see Figure 1). It is guaranteed that if two or more processes execute the splitter, then there are two processes which obtain different values. Thus the set of the processes is “split” into smaller subsets, according to the values obtained.

The implementation of the splitter presented in [17] uses two shared registers, $X$ and $Y$. Initially, $X = -$ and $Y = false$. Processes $p_i$ executing the splitter first writes $id_i$ into $X$. After that $p_i$ reads $Y$. If $Y = true$ then $p_i$ obtains right. Otherwise, the process sets $Y = true$ and checks the contents of $X$. If $X$ still contains $id_i$, then $p_i$ obtains stop, otherwise ($X \neq id_i$) $p_i$ obtains left.

We arrange the splitters in a complete binary tree with depth $n - 1$ (see Figure 2). A process starts registration at the root of the tree, and moves down the tree (either left or right).
Consider all the executions \( \alpha \) such that \( k(\alpha) \leq K \), for some \( K \leq n \). For these executions, \( K \) is the upper bound on the number of active processes.

An algorithm is \textit{fast} if in executions with at most \( K \) active processes its step complexity is bounded by a function of \( K \). We suppose that parameter \( K \) is fixed and known in advance, and, therefore, it can be used by a fast algorithm explicitly.

An algorithm is \textit{adaptive} if its step complexity in any execution \( \alpha \) is bounded by a function of the actual number of active processes in \( \alpha \), \( k(\alpha) \). Since \( k(\alpha) \) depends on the specific execution, adaptive algorithm has no \textit{a priori} information of the degree of contention in the system.

### 2.2 Definitions of the Problems

**The Collect Problem.** A solution for the \textit{collect} problem should define algorithms for two operations — \textit{registration} and \textit{collect}. Each active process first declares that it participates in the computation by performing \textit{registration}. After that, a process \( p_i \) may perform \textit{collect} arbitrary many times. Operation \textit{collect} returns a subset of the active processes, called a \textit{view}, \( V_i \).

Intuitively, the returned view should contain a subset of the active processes which perform \textit{registration} before or concurrently with the \( p_i \)'s execution of \textit{collect}. The following conditions must hold, for any processes \( p_i \) and \( p_j \) (not necessary distinct):

1. If \( p_j \) completes the \textit{registration} before \( p_i \) starts a \textit{collect} returning \( V_i \), then \( p_j \in V_i \).

2. If \( p_j \) starts the \textit{registration} after \( p_i \) completes a \textit{collect} returning \( V_i \), then \( p_j \notin V_i \).

3. If \( p_i \) completes a \textit{collect} returning \( V_i \) before \( p_j \) starts a \textit{collect} returning \( V_j \), then \( V_i \subseteq V_j \).

**The Lattice Agreement Problem.** For the \textit{lattice agreement} problem we use the following definition, which is equivalent to the definition given in [6]. A process \( p_i \) starts with \( V_{in}(p_i) = \{ p_i \} \) and is required to decide on a subset of the active processes (e. q. \textit{view}), \( V_{out}(p_i) \), such that the following conditions hold:

**Comparability:** For any \( i \) and \( j \), either \( V_{out}(p_i) \subseteq V_{out}(p_j) \) or \( V_{out}(p_j) \subseteq V_{out}(p_i) \).

**Self-containment:** For any \( i \), \( V_{in}(p_i) \subseteq V_{out}(p_i) \).

**The M-Renaming Problem.** The \( M \)-renaming problem is defined as follows. Each of the \( n \) processes has a unique name in the range \( \{0 \ldots N-1\} \). The processes are required to choose distinct names in a range \( \{0 \ldots M-1\} \), where \( M < N \). The range \( \{0 \ldots N-1\} \) is the \textit{initial name space}, and the range \( \{0 \ldots M-1\} \) is the \textit{final name space}. 

7
of \( R \), and the response to the operation is the current state (content) of \( R \). The \textit{write}(\( v, R \)) operation changes the state of \( R \) to \( v \), and the response of the operation is \textit{ack}.

There are several types of read/write registers which vary in the number of processes that are allowed to perform reads and writes on the same register. A \textit{multi-writer multi-reader} register allows any number of processes to perform \textit{read} and \textit{write} operations. A \textit{single-writer multi-reader} register allows only one process to perform \textit{write} operations, and any number of processes to perform \textit{read} operation. A single-writer multi-reader register is \textit{static} if the identity of the process which can write into the register is fixed before the execution starts, and this identity does not change from execution to execution. A single-writer multi-reader register is \textit{dynamic} if the identity of the process which can write into the register depends on the execution itself and may vary from execution to execution.

**Wait-free asynchronous computation.** We consider totally asynchronous shared memory systems. The processes in such systems do not have access to global or local clocks of any kind. Each process runs at an arbitrary speed, and there are no bounds on the variation of speed among the processes.

In addition, processes in the system may be subjected to \textit{crash failures}. A process that fails by crashing stops to execute its code and never recovers. It is assumed that any number of processes may fail during the execution. An algorithm which guarantees that each non-faulty process successfully completes its computation in a finite number of steps is called \textit{wait-free}.

Communication in wait-free algorithms have a unique property: Since all the processes may crash, a process' code can never depend on the behavior of other processes. In other words, each process must be able to complete its computation in a finite number of steps, regardless of operations taken by other processes. In particular, a process can never perform any kind of \textit{wait} operation, since if all other processes fail, the process will wait forever. (Process can not use time out to stop waiting, since it can not access any kind of clock.)

**The complexity measure.** We assume that the main part of the computation time is consumed by the operations on shared primitives, and not by the internal operations of the processes. Thus, in the step complexity of an algorithm we count only reads and writes of shared registers, performed by a process during the execution.

Let \( \alpha \) be an execution of some algorithm \( A \). Denote by \( \text{step}(A, \alpha, p_i) \) the number of read/write operations on shared registers performed by a process \( p_i \) in \( \alpha \). The step complexity of algorithm \( A \) is the maximum of \( \text{step}(A, \alpha, p_i) \), taken over all executions \( \alpha \) and all processes \( p_i \).

In this work we consider two types of algorithms whose step complexity does not depend on the total number of the processes in the system — \textit{fast} and \textit{adaptive} algorithms.

A process is \textit{active} in execution \( \alpha \) if it takes steps in \( \alpha \). Denote by \( k(\alpha) \) the number of active processes in \( \alpha \).
2 Preliminaries

2.1 The Computation Model

Throughout the work, we assume the wait-free asynchronous shared-memory model. The following description of the model’s concepts and components is based on [20, 13].

Processes and shared primitives. A shared memory system consists of two basic types of components, processes and shared primitives.

A process is described as an I/O automaton [16]; it has a set of possible states, a set of initial states, and a state transition function. The automaton executes a thread of sequential code step by step. The operations of the process may be either “internal” or “external”. An internal operation is an operation in which the process changes its state only according to its current state. This type of operations represents local computations of the process. In an external operation, the process invokes a request to some shared primitive in the system, and changes its state according to the response from that primitive. This kind of operations are the input/output events of the the processes, which represent the communication of the individual process with the rest of the system.

A shared primitive is a data structure that enables the processes in the system to invoke requests and receives responses. A shared primitive can be described as an I/O automaton as well. It also has a set of states, a set of possible initial states, and a transition function. However, instead of executing a thread of local code, the automaton representing a shared primitive simply waits for requests. Any request defines an input event that causes the automaton to change its current state and to respond to the request by issuing an output event.

The execution of the shared memory system is described as a series of pairs \((request, response)\). Each pair represents a process invoking a request to some shared primitive, and the response from that primitive to the process. Notice that the internal operations of the processes do not appear in the execution. Intuitively, this means that interleaving of the processes’ internal operations has no effect on the global behavior of the system.

An important property of the execution described above is that it does not separate the \((request, response)\) pairs. That is, the invocation of the request and its response are assumed to occur “at the same time”. This kind of executions reflect the intuitive notion that the operations on the system’s primitives are atomic.

Atomic read/write registers. In this work we consider shared primitives called atomic read/write registers.

The data structure that represents a read/write register is a memory cell, and the state of the register is the content of the cell. The requests, or operations, that processes can invoke on register \(R\) are \(\text{read}(R)\) and \(\text{write}(v, R)\). The \(\text{read}(R)\) operation does not change the state
Anderson and Moir [3] observe that a renaming algorithm can improve the performance of computations whose complexity depends on the size of the processes' name space, $N$. If processes first acquire new names in a smaller range, then the complexity of the computation can be made independent of the size of the initial name space. However, the complexity of the renaming algorithms presented so far depends on $N$, thus the overall complexity of the computation also remains dependent on $N$.

This observation motivated design of renaming algorithms whose complexity does not depend on the size of the initial name space. Moir and Anderson [17] suggest that a renaming algorithm be called *fast* if its step complexity depends only on $K$, the maximal number of participating processes. They present fast algorithm for both one-time and long-lived renaming. However, the algorithms use strong synchronization primitives, such as *test-and-set*.

Moir and Garay [18] introduce a fast $O(k)$ algorithm for one-time $k(K + 1)/2$-renaming, using only *read* and *write* operations. By combining the fast $k(K + 1)/2$-renaming algorithm and the $(2k - 1)$-renaming algorithm of Burns and Paterson [10], Moir and Garay obtain a fast $O(K^3k)$ algorithm for one-time $(2k - 1)$-renaming.

The *lattice agreement* problem was introduced by Attiya, Herlihy and Rachman [6]. They show a transformation of any lattice agreement algorithm into an implementation of atomic snapshot. The transformation requires $O(n)$ additional read/write operations. Using this transformation, Attiya, Herlihy and Rachman [6] construct a randomized implementation of atomic snapshot object, which requires an expected number of $O(n)$ read/write operations.

Attiya and Rachman [7] present a deterministic algorithm for lattice agreement that requires $O(n \log n)$ operations on static single-writer multi-reader registers. The algorithm can be transformed into an implementation of atomic snapshots with step complexity $O(n \log n)$, which uses single-writer multi-reader registers.

Inoue, Chen, Masuzawa and Tokura [15] introduce a linear-time algorithm for lattice agreement using *multi-writer multi-reader* registers. This algorithm yields an implementation of atomic snapshots which requires $O(n)$ operations on multi-writer multi-reader registers for each operation on the object.

Choy and Singh [11] present adaptive solutions for the *mutual exclusion problem*, using only read/write operations. Their algorithms are adaptive with respect to the *amortized* step complexity; in the worst case, the step complexity of the algorithms depends on $n$. In addition, the algorithms are not wait-free.

Afek, Dauber and Touitou [2] introduce universal methods for adaptive implementations of long-lived objects (which they call fast, by mistake). The step complexity of the implementations depends linearly on the actual number of processes that access the object concurrently; however, they require strong *load-linked* and *store-conditional* operations.
- $O(K \log K)$ fast algorithm for one-time $(2k-1)$-renaming.

The step complexity of the best existing algorithm for fast $(2k-1)$-renaming is $O(K^3k)$ [18].

- $O(N)$ algorithm for one-time $(2k-1)$-renaming.

The best existing algorithm which reduces the size of the name space from $N$ to $2k-1$ has step complexity $O(NK^2)$ [10].

- $O(n)$ algorithm for lattice agreement which uses only dynamic single-writer single-reader registers. This algorithm yields an $O(n)$ implementation of atomic snapshots using dynamic single-writer multi-reader registers.

Previously, there were an $O(n \log n)$ implementation of atomic snapshot which requires static single-writer multi-reader registers [7], and an $O(n)$ implementation which requires multi-writer multi-reader registers [15].

The size of the name space provided by our fast renaming algorithm is optimal, since wait-free $M$-renaming is possible with reads and writes only if $M \geq 2k-1$ [14]. Although the size of the name space provided by our adaptive renaming algorithm is not optimal ($M = 6k-1$), it is still linear in the number of participating processes.

1.3 Historical Perspective

The renaming problem was introduced by Attiya, Bar-Noy, Dolev, Peleg and Reischuk for message-passing systems [5]. They present an algorithm with exponential step complexity for $(n+t)$-renaming, where $t$ is the maximal number of faulty processes.

Bar-Noy and Dolev [8] present a wait-free algorithm for renaming in the asynchronous shared-memory model. Their solution achieves a name-space of size $2n-1$ and requires an exponential number of read/write operations.

Long-lived renaming was first solved by Burns and Peterson [10]. They consider the $l$-assignment problem, which requires dynamic allocation of $l$ distinct resources to $n$ processes, assuming that at most $t$ of them fail. Burns and Peterson present an algorithm which works correctly whenever $l \geq 2t-1$; the step complexity of the algorithm is $O(Nk^3)$. In the wait-free case, when all active processes may fail ($t = k - 1$), their algorithm solves the long-lived renaming problem with name space of size $2k-1$.

Burns and Peterson [10] prove that long-lived $M$-renaming can not be solved in a wait-free manner if $M < 2k-1$. Herlihy and Shavit [14] show that one-time renaming also requires $2k-1$ names. This implies that the renaming algorithms mentioned above have optimal name space.

Borowsky and Gafni [9] present algorithm which solves $(2k-1)$-renaming using $O(N^2k)$ read/write operations.
Although fast algorithms provide good performance in some cases, their advantage is quite restricted in more general situations. The main shortcoming of fast algorithms is that they rely on \textit{a priori} knowledge of \( K \), the maximal number of active processes. It is often difficult to determine \( K \) before execution; moreover, the complexity of fast algorithms is not optimal when actual contention is much lower than the upper bound, \( K \).

The notion of an \textit{adaptive} algorithm is intended to overcome the shortcomings of fast algorithms. Choy and Singh [11] suggest to call an algorithm \textit{adaptive} if its complexity depends only on the actual number of active processes, \( k \); \( k \) need not be fixed in advance and it may change in different executions of the algorithm. This definition guarantees that the complexity of an adaptive algorithm adjusts to the degree of contention in the system: it is constant if a single process executes the algorithm, and it grows gradually as the number of active processes increases.

### 1.2 Contributions of this Work

The main results of this work are adaptive wait-free algorithms for the \textit{collect} problem [4], the \textit{lattice agreement} problem [6] and the \textit{renaming} problem [5] in the shared-memory model with read/write operations. Besides that, we improve the step complexity of the existing (non-adaptive) algorithms for lattice agreement and renaming.

Below we describe briefly the problems addressed in the work; more formal definitions appear in Section 2.

In the \textit{collect} problem, a process is required to collect an up-to-date set of active processes. Since this operation is widely used, an adaptive solution for the collect problem can improve the step complexity of many existing algorithms [1, 7, 9, 10, 12].

In the \textit{lattice agreement} problem, each process starts with an element of some partially ordered lattice; processes are required to decide (in a non-trivial manner) on elements which are \textit{comparable} in the lattice. \textit{Atomic snapshot object} allows to processes to get an instantaneous global view (“snapshot”) of the shared memory, and is useful for simplifying the design of wait-free algorithms. A wait-free lattice agreement algorithm can be turned into a wait-free implementation of atomic snapshots, with \( O(n) \) additional read/write operations [6].

In the \textit{M-renaming} problem [5], each process starts with a name in the range \( \{0 \ldots N - 1\} \) and is required to choose a distinct name in a smaller range, \( \{0 \ldots M - 1\} \), for some \( M < N \). This is a special case of the \textit{long-lived M-renaming} problem, in which processes repeatedly \textit{acquire} and \textit{release} names from a range \( \{0, \ldots, M - 1\} \), for some \( M < N \). In this work we consider only the one-time version of the renaming problem.

The main contributions of this work are an adaptive \( O(k) \) collect algorithm, an adaptive \( O(k \log k) (6k-1) \)-renaming algorithm and an adaptive \( O(k \log k) \) lattice agreement algorithm.

These algorithms are designed using a number of components, some of which are interesting on its own:
1 Introduction

An asynchronous shared-memory system consists of a set of processes that communicate by performing read and write operations on registers that reside in the shared memory. The processes are completely asynchronous and do not have access to global or local clocks of any kind. Each process runs at an arbitrary speed, and there are no bounds on the variation of speed among the processes.

We consider wait-free algorithms, which are guaranteed to execute correctly even if any number of processes fail by crashing during the execution.

Many standard synchronization techniques of distributed programming are not suitable for wait-free algorithms. Since there are no assumptions about the relative speeds of processes, a non-faulty process can not distinguish between crashed and delayed processes. Therefore, wait-free algorithms can not depend on any type of wait operations. Similarly, wait-free algorithm can not use critical sections, since a process crashed in a critical section may prevent other processes from making progress.

1.1 Algorithms that Adapt to Contention

In this work, we focus on the step complexity of wait-free algorithms, which is measured by the maximal number of shared memory operations performed by a process during the algorithm.

Most of the existing wait-free algorithms for the shared-memory model with read/write operations have step complexity which depends on the total number of the processes in the system, $n$. That is, the expression for step complexity contains a term $f(n)$, for some function $f$.

In real distributed systems, the total number of processes may be very large, while most of the time only a small subset of the processes participate in the same algorithm. In these settings, step complexity which depends on the total number of the processes does not seem optimal.

This work studies wait-free algorithms whose complexity adjusts to the degree of contention in the system. The step complexity expression for these algorithms does not contain any terms of the form $f(n)$, for any function $f$. Instead of that, the complexity depends only on the number of processes that participate in the algorithm (the active processes).

Particularly, we consider two types of algorithms with contention-dependent complexity — fast algorithms and adaptive algorithms.

As defined by Moir and Anderson [17], an algorithm is fast if its step complexity depends only on $K$, the maximal number of active processes, but not on the total number of processes, $n$. The parameter $K$ should be set in advance and it can not change from one execution to another.
Abstract

In shared-memory distributed systems, $n$ independent asynchronous processes with distinct names in the range $\{0, \ldots, N-1\}$, $n \leq N$, communicate by reading and writing to shared memory. In such systems, processes may operate at very different speeds or fail entirely. *Wait free algorithms* have been proposed for fault-tolerant computations in the presence of an arbitrary number of crash failures and different processes’ speed. A wait-free algorithm guarantees that each non-faulty process continues its correct execution regardless of the behavior of other processes.

The complexity of *fast* and *adaptive* algorithms considered in this work adjusts to the level of contention in the system. An algorithm is *fast* if its complexity depends on the maximal number, $K$, of active processes which is known in advance, but not on the total number, $n$, of processes in the system. An algorithm is *adaptive* if its complexity depends only on the actual number, $k$, of active processes which is unknown in advance and may change in different executions of the algorithm.

It is shown that two important decision problems, *lattice agreement* and *renaming with a linear name space* have adaptive solutions using only read and write operations. The main results of this thesis are an adaptive algorithm for lattice agreement with step complexity $O(k \log k)$ and an adaptive algorithm for $(6k-1)$-renaming with step complexity $O(k \log k)$.

These algorithms are constructed using a number of components, some of which are interesting on their own:

- $O(K \log K)$ fast algorithm for $(2k-1)$-renaming. The best existing fast algorithm for $(2k-1)$-renaming has step complexity $O(K^2k)$.
- $O(N)$ algorithm for one-time $(2k-1)$-renaming. The best existing algorithm which reduces the size of the name space from $N$ to $2k-1$ has step complexity $O(Nk^3)$.
- $O(n)$ algorithm for lattice agreement which uses only dynamic single-writer single-reader registers.

The last algorithm yields an $O(n)$ implementation of atomic snapshots using dynamic single-writer multi-reader registers. The best existing implementations of atomic snapshots are an $O(n \log n)$ implementation using static single-writer multi-reader registers, and an $O(n)$ implementation using multi-writer multi-reader registers.
Adaptive Wait-free Algorithms for Lattice Agreement and Renaming

Hagit Attiya and Arie Fouren
Department of Computer Science
The Technion, Haifa 32000, Israel

April 1998
Adaptive Wait-free Algorithms
for Lattice Agreement and Renaming

by

Hagit Attiya and Arie Fouren

Technical Report #0931
April 1998

Most of the Technical Reports of the CS Dept. may be accessed on line through the depart-
ment's Home Page. The address is: http://www.cs.technion.ac.il/Reports/

For the elimination of any doubt, it is hereby stressed that the staff member and/or the
Technion and/or the Technion Research and Development Foundation Ltd. will not be liable
for any property damage and/or corporeal damage and/or expense and/or loss of any kind
or sort that will be caused or may be caused to you or to anyone acting on your behalf, in
consequence of this statement of opinion or this report, or in any connection to it.