Adaptive and Efficient Mutual Exclusion*

[Extended Abstract]

Hagit Attiya and Vita Bortnikov
Department of Computer Science
The Technion
Haifa 32000, Israel
hagit@cs.technion.ac.il
vitab@cs.technion.ac.il

ABSTRACT

A distributed algorithm is *adaptive* if its performance depends on k, the number of processes that are concurrently active during the algorithm execution (rather than on n, the total number of processes). This paper presents adaptive algorithm for mutual exclusion using only read and write operations.

The worst case *step complexity* cannot be a measure for the performance of mutual exclusion algorithms, because it is always unbounded in the presence of contention. Therefore, a number of different parameters are used to measure the algorithm's performance: The *remote step complexity* is the maximal number of steps performed by a process where a *wait* is counted as one step. The *system response time* is the time interval between subsequent entries to the critical section, where one time unit is the minimal interval in which every active process performs at least one step.

The algorithm presented here has $O(k)$ remote step complexity and $O(\log k)$ system response time, where k is the point contention. The space complexity of this algorithm is $O(nN)$, where N is the range of processes' names.

The space complexity of all previously known adaptive algorithms for various long-lived problems depends on N. We present a technique that reduces the space complexity of our algorithm to be a function of n, while preserving the other performance measures of the algorithm.

1. INTRODUCTION

The *mutual exclusion* problem is to design a protocol that guarantees mutually exclusive access to a critical section among competing processes. This problem has been studied for many years, dating back to the seminal paper of Dijkstra [13]. Most of the published solutions to the mutual exclusion problem require an incoming process to look at every other potential competitor as a part of its entry code. Recently it has been observed that the worst case complexity of distributed algorithms could be adaptive, that is, bounded by a function of the number of actually *active* processes, which can be very small.

Lamport [16] suggested a mutual exclusion algorithm which requires a constant number of steps when a process runs alone and an unbounded number of steps if two or more processes run concurrently. Alur and Taubenfeld [5] showed that for any asynchronous mutual exclusion algorithm there is no bound on the number of shared-memory operations taken by the winning process in the presence of contention. Thus, the *step complexity* of any mutual exclusion could not be adaptive. Following this, a number of different parameters were suggested to measure the effectiveness of mutual exclusion algorithms. The *remote step complexity* [11, 17, 20] is the maximal number of shared memory operations performed by a process, where a *wait* is counted as a single operation (this parameter is well-defined only for lockout-free algorithms). The number of *remote memory references* [21] is a stronger version of this parameter. It assumes a model where each shared location is local to a single process and remote for all other processes, and counts the number of remote memory references, assuming process spins only on local locations. The *system response time* [11] is the time interval between subsequent entries into the critical section, where a time unit is the minimal interval of time in which every active process performs at least one step.

An algorithm is *adaptive* if its complexity is bounded by a function of the number of contending processes, denoted k; k is unknown in advance, and it may change in different executions of the algorithm. The strongest form of adaptiveness requires the complexity of an operation to be bounded by a function of its *point contention*, defined as the maximum number of processes executing concurrently at some point during the operation's interval.

Our basic algorithm has $O(k)$ remote step complexity and its system response time is $O(\log k)$, where k is point contention. The algorithm is constructed from an adaptive long-lived *non-wait-free* k-renaming and an adaptive *tourna-*

*This research was supported by the fund for the promotion of research in the Technion.

PODC 2000 Portland Oregon

Algorithms	Remote Step Complexity	System Response Time	Space Complexity
Choy and Singh [11]	$O(N)$	$O(k)$	$O(N)$
Adaptive Bakery Algorithm [4]	$O(k^4)$	$O(k^4)$	$O(N^3)$
Afek et al. [3]	$O(\min(k^2, k \log N))$	$O(k^2)$	$O(N2^{2n})$
Anderson and Kim [7]	$O(k)$	$O(k)$	$O(N)$
First Algorithm	$O(k)$	$O(\log k)$	$O(nN)$
Second Algorithm	$O(k)$	$O(\log k)$	$O(n^2)$

Table 1: Comparison with previous adaptive mutual exclusion algorithms

ment tree for mutual exclusion. Our non-wait-free renaming algorithm has a much better remote step complexity and system response time than all known adaptive long-lived wait-free renaming algorithms.

Arguing about the point contention in the complexity proofs of our algorithms requires novel proof techniques. Proofs of this style appear in [1, 2, 4], and resemble the potential method used in amortized analysis.

The space complexity of this algorithm is $O(nN)$. We present a technique to make the space complexity depend solely on n. This technique achieves $O(n^2)$ space complexity, however the remote step complexity of the first entry to the critical section of every process increases to $O(k')$, where k' is the operation's *interval contention* - the total number of processes that are active during the operation interval.

A number of mutual exclusion algorithms with $o(N)$ time complexity were designed.

Choy and Singh [11] presented an adaptive mutual exclusion algorithm with $O(k)$ system response time and $O(N)$ remote step complexity. The amortized system response time of their algorithm is $O(1)$.

Yang and Anderson [21] presented an algorithm that uses a tournament tree, where in each node a two-process mutual exclusion algorithm is located. Their algorithm induces $O(\log N)$ remote memory references; even if there is no contention, $\Theta(\log N)$ steps should be performed by a process to enter the critical section. Anderson and Kim [6, 7] presented an algorithm with $O(1)$ remote memory references in the absence of contention and $O(\log N)$ under contention. They also mention [7] an algorithm with $O(k)$ remote memory references.

Afek et al. [4] demonstrated how adaptive long-lived collect can be used to transform the Bakery algorithm [15] into an adaptive mutual exclusion algorithm. Since adaptive long-lived collect has $O(k^4)$ step complexity, the remote step complexity of the resulting mutual exclusion algorithm is also $O(k^4)$. They present another adaptive mutual exclusion algorithm [3]; both the system response time and the remote step complexity of this algorithm are $O(k^2)$.

Table 1 summarizes the results of this paper and compares them to the known adaptive mutual exclusion algorithms.

2. PRELIMINARIES

We assume a standard asynchronous shared-memory model of computation [14]. A system consists of n processes, $p_1, \ldots, p_n$, communicating by reading and writing to shared registers. Each process can read from and write to any register (*multi-writer multi-reader* registers).

A process participating in the mutual exclusion algorithm loops through the following sections: *entry* (enter procedure), *critical*, *exit* (exit procedure) and *remainder*.

Let α be an execution of a mutual exclusion algorithm A; let α' be a finite prefix of α.

Process p_i is *active* at the end of α' if α' includes an invocation of enter by p_i without a return from the matching exit. $Cont(\alpha')$ is the set of active processes at the end of α'. The *point contention* at the end of α' is $|Cont(\alpha')|$, denoted $PntCont(\alpha')$.

Consider a finite execution interval β of α; we can write $\alpha = \alpha_1\beta\alpha_2$. The point contention during β, denoted $PntCont(\beta)$, is the maximum point contention over all prefixes $\alpha_1\beta'$ of $\alpha_1\beta$. If the point contention during β is k, then for some prefix β' of β, $PntCont(\alpha_1\beta') = k$.

The *interval contention* of β, denoted $IntCont(\beta)$, is the total number of different processes that are active during the operation interval. Clearly, for any interval β, $PntCont(\beta) \leq IntCont(\beta)$ and the interval contention of β is bounded by n, the total number of processes.

The *remote step complexity* of process p_i during β is the number of steps performed by p_i in β, when a *wait* operation is counted as one step. The remote step complexity of a mutual exclusion algorithm is *adaptive* (*to point contention*) if there is a bounded function S, such that the remote step complexity of any process p_i in an execution interval of enter, β, and the matching exit is at most $S(PntCont(\beta))$.

The *time complexity* of β is the number of time units during β, where one time unit is the minimal execution interval in which each active process performs at least one step. An algorithm has *adaptive* (*to point contention*) *system response time*, if there is a bounded function T, such that the time complexity of any interval β between two subsequent critical section executions is at most $T(PntCont(\beta))$.

3. THE BASIC ALGORITHM

In this section we present a basic algorithm using an unbounded memory. A technique to bound the memory is described in Section 4. In the algorithm, a process gets a name in a range of size $O(k)$ using *long-lived renaming* [8, 18], and uses this name to enter an adaptive tournament tree for mutual exclusion. The winner of the tournament tree enters the critical section.

Many long-lived renaming algorithms are known [1, 2, 10, 18], and some of them are adaptive [1, 2]. Unfortunately, using any of these algorithms drives for very high complexity. In the context of mutual exclusion, the complexity of renaming could be significantly improved, since wait-freedom is not required.

3.1 Non-Wait-Free Renaming

In the *long-lived M-renaming* problem, processes repeatedly acquire and release distinct names in the range $\{1, \ldots, M\}$.

Our algorithm uses an array of n entries. Each entry contains a pointer to a *chain of filters* (or simply, a *chain*). A process tries to win in the chains, one after the other, until successful in some chain. The name that the process receives is the index of the chain it wins.

In a chain, filters are concatenated one after the other. A process that enters a filter leaves it by receiving either *success* or *fail*. Only a process that succeeds in a filter proceeds to the next filter in the chain or concludes that it is the winner of the chain. A process that fails in a filter, loses in the current chain and moves to the following chains. If the process failed in the r'th filter of the l'th chain, then it skips the next $r-1$ chains and tries to win in the $(l+r)$'th chain.

We show below that if one or more processes enter a chain, then exactly one process wins it. Moreover, if some process fails in the r'th filter in a chain, then there are processes that failed in filters $0 \ldots r-1$ of the chain. We prove that process p_i accesses the l'th chain only if there are at least l processes which are simultaneously active at some point during p_i's enter operation. We will also argue that the winner of the chain is determined in $O(\log k)$ time units, where k is the point contention during the winner's execution interval.

3.1.1 The Code

The pseudocode appears in Algorithm 1. Chains are stored in an array *Chains*. There is an infinite number of filters in each chain, numbered $0, 1, \ldots$. An additional shared data structure is an array *startFilter*$[0, \ldots, n-1]$; *startFilter*$[l]$ contains the index of the current entry filter of *Chains*$[l]$.

Procedure getName, obtaining a new name, invokes procedure executeChain for executing a chain with a chain's index as parameter. executeChain returns a pair: *win* or *lose* indicating whether the processes wins or loses the chain, and the index of the last accessed filter in the chain.

Our filter is a modification of the filter of Choy and Singh [11]. Their filter provides the following properties:

Safety: If k processes enter the filter, then no more than $\lceil \frac{k}{2} \rceil$ processes succeed in it.

Progress: If one or more processes enter the filter, then at least one process succeeds in it.

Algorithm 1 Adaptive long-lived non-wait-free k-renaming

```
private variables
    lastFilter : integer

procedure getName()                          // get a new name
1:  l := 0
2:  index := −1
3:  repeat
4:      l := l + index + 1
5:      ⟨result, index⟩ := executeChain(l)
6:  until result = win
7:  lastFilter := startFilter[l] + index
8:  return l

procedure releaseName(name)                  // release a name
1:  startFilter[name] := lastFilter + 1

procedure executeChain(l)                    // access chain l
1:  Filters := Chains[l][startFilter[l]]
                               // move the pointer to the start
2:  curr := 0
3:  while (true)
4:      if executeFilter(Filters[curr]) = success then
5:          if curr > 0 and ¬Filters[curr − 1].c then
6:              return ⟨win, curr⟩
7:          else curr++          // proceed to the next filter
8:      else return ⟨lose, curr⟩

procedure executeFilter(filter)    // access a specific filter
1:  if filter.turn ≠⊥ then return fail
2:  filter.turn := id
3:  if filter.d then return fail
4:  wait until ¬filter.b or filter.turn ≠ id or filter.d
5:  if filter.d then return fail
6:  if filter.turn = id then filter.b := true
7:  if filter.turn ≠ id then           // failing in the filter
8:      filter.c := true
9:      filter.b := false
10:     return fail
11: else                            // succeeding in the filter
12:     filter.d := true
13:     return success
```

We upgrade the filter so it also have the following property:

Time complexity: Some process succeeds in the filter $O(1)$ time units after the first process enters it.

The major difference between the code of our filter and the code of Choy and Singh filter is the second condition in Line 4. If a process determines that *turn* $\neq$ *id* (Line 4), then this process will never succeed in the filter, and therefore it fails at this point (and does not continue to wait for $\neg b$).

3.1.2 Correctness Proof

The proofs of the safety and progress properties of the filter are similar (although not identical) to the proofs in [11]; they are postponed to the full version of the paper.

We say that a process *enters* a filter if it passes Line 1 of executeFilter. The first two lines of executeFilter ensure that the last process enters the filter $O(1)$ time units after the first process enters it. In addition, b becomes $false$ $O(1)$ time units after $turn$ is set by the last entering process. This implies the time complexity property of the filter.

LEMMA 3.1 (TIME COMPLEXITY). *Some process succeeds in the filter $O(1)$ time units after the first process enters it.*

The safety and progress properties of the filter ensure that exactly one process is eventually left in the chain and wins it. Formally, process p_i *wins* chain l if it returns win from executeChain(l). We refer to p_i as a *winner* of chain l even before it actually wins.

A process that is not in the remainder section, is *accessing* the last filter it called executeFilter for, and the chain this filter belongs to. Chain l is *busy* if $Chains[l][startFilter[l]].turn \neq \perp$ and *empty* otherwise; that is, the chain's winner is accessing it.

For chain l, an execution is partitioned into *rounds*. The first round starts at the beginning of the execution; a new round starts when chain l becomes empty. A winner is accessing the chain until the end of the round. Filters in chain l are counted relative to the value of $startFilter[l]$ at the beginning of the round, where the first filter is counted 0. A process *enters* chain l if it enters the first filter of the current round of chain l. In the following, we refer to a single round of a chain, unless specified otherwise.

LEMMA 3.2. *If some process enters a chain, then exactly one process wins it.*

SKETCH OF PROOF. By the progress property, if some process enters a filter then at least one process succeeds in it. By the algorithm, the process that succeeds in $Filters[r]$ either wins the chain or enters $Filters[r+1]$. Hence, in any execution of the algorithm some process either wins the chain or enters $Filters[\lceil \log k \rceil + 1]$. By the safety property, at most one process enters $Filters[\lceil \log k \rceil]$; this process succeeds in $Filters[\lceil \log k \rceil + 1]$, finds $Filters[\lceil \log k \rceil].c = false$ and wins the chain. Hence, some process wins the chain.

When a winner executes Line 5 and discovers that $Filters[curr-1].c = false$, no process has failed in $Filters[curr-1]$. Therefore, no other process is currently in $Filters[curr]$. The winner checks $Filters[curr-1].c$ after setting $Filters[curr].d$ to $true$. Hence, every process that enters $Filters[curr]$ from this point on, returns $fail$ in Line 3 of executeFilter and does not succeed in this filter. Therefore, there is a single winner. □

The last lemma implies that no two processes have the same name at the end of an execution prefix.

An *execution interval* of a process includes one iteration of enter, critical section, and exit.

All processes entering a filter read $turn$ in Line 1 of executeFilter before the first entering process writes its id to $turn$ in Line 2. Thus, all these processes are active at the first write of $turn$. Therefore, we have the following lemma:

LEMMA 3.3. *If some process enters a filter, then there is a point in its execution interval in which all processes entering this filter are active.*

Therefore, if k processes enter a filter, the point contention during execution iterval of each entering process is at least k. From the chain definition, if k processes enter a chain, then the point contention of the winner's execution interval is at least k. By the safety property of a filter, at most one process enters $Filters[\lceil \log k \rceil]$; this process suceeds in $Filters[\lceil \log k \rceil + 1]$, finds $Filters[\lceil \log k \rceil].c = false$ and wins the chain. This implies the following lemma:

LEMMA 3.4. *If k processes enter the chain, then some process wins the chain in $\lceil \log k \rceil + 2$ filters.*

From the last two lemmas we conclude that $O(\log k)$ filters are required to determine the winner in a chain, where k is the point contention of the winner's execution interval. By the time complexity property of the filter, after $O(1)$ time units there is a process that succeeds in a single filter. Therefore, we have the following lemma:

LEMMA 3.5. *Some process wins the chain within $O(\log k)$ time units after the chain became busy, where k is the point contention of the winner's execution interval.*

Thus, the system response time of our renaming algorithm is $O(\log k)$. The following lemmas lead to the remote step complexity of the algorithm.

LEMMA 3.6. *If some process p_i fails in $Filters[r]$ of some chain l, then there are processes $p_{m_0}, \ldots, p_{m_{r-1}}$ that enter and fail in $Filters[0], \ldots, Filters[r-1]$ of chain l, respectively.*

PROOF. We prove the lemma by induction on r. The base case, $r = 0$ is trivial.

For the induction step, assume the lemma holds for $Filters[r]$, and consider $Filters[r+1]$. By the progress property, if p_i fails in $Filters[r+1]$, then there is another process p_j that succeeds in $Filters[r+1]$. By the algorithm, both p_i and p_j succeed in $Filters[r]$. Therefore, by the safety property, at least one process p_{m_r} enters and fails in $Filters[r]$. Thus, by the induction hypothesis, there are processes $p_{m_0}, \ldots, p_{m_{r-1}}$ that enter and fail in $Filters[0], \ldots, Filters[r-1]$, respectively. Process p_{m_r} is not one of $p_{m_0}, \ldots, p_{m_{r-1}}$, since it enters $Filters[r]$, and therefore succeeds in $Filters[0], \ldots, Filters[r-1]$. Thus, there are processes $p_{m_0}, \ldots, p_{m_{r-1}}, p_{m_r}$ that enter and fail in $Filters[0], \ldots, Filters[r-1], Filters[r]$, respectively. □

The following lemmas argue about the entire execution of a chain and not a single round of it. For a finite execution prefix α', the current round of chain l is the round of chain l at the end of α'. Whenever we refer to filter r of a chain at the end of α', we mean the r'th filter of the current round of this chain.

Consider process p_i accessing the r'th filter of chain l at the end of α'. The *location* of p_i at α' is defined as follows: if p_i is the winner of the current round of chain l then its location is l otherwise its location is $l+r+1$. The location of p_i is updated by executing Line 2 of executeFilter, if it enters the filter and by executing Line 1 of executeFilter, otherwise. This implies that p_i's location is incremented by at most 1 in one step of p_i.

The *location* of filter r of chain l is $l+r+1$.

Let α' be a prefix of α in which no process has location s, and let α'' be the longest prefix of α' in which chain s is busy. By the definition of location, chain s is empty at α'. Let β be the interval between α'' and α', that is $\alpha' = \alpha''\beta$. Note that chain s is empty in β.

LEMMA 3.7. *The number of processes with locations $s, \ldots, \infty$ at the end of α' is at most the number of processes with locations $s, \ldots, \infty$ at the end of α''.*

PROOF. By the definition of β, there is no process with location s at the end of β. Therefore, any process that changes its location from $s-1$ to s in β has location $\geq s+1$ at the end of β. Assume that process p_i has location $\leq s-1$ at the beginning of β and has location $\geq s+1$ at the end of β. Consider the possible ways for p_i to change its location from s to $s+1$:

Case 1: p_i fails in some filter with location s. Thus, p_i changes its location to $s+1$ only when it accesses the first filter of chain s, but it is not the winner of the chain. However, when p_i updates its location, $turn \neq \perp$ for the first filter of chain s, contradicting the fact that chain s is empty during β.

Case 2: p_i succeeds in some filter with location s and enters a filter with location $s+1$. By p_i's definition, its location is $< s$ at the beginning of β. Therefore, it enters some filter with location s and succeeds in it in β. However, by the fact that p_i is not the winner of chain s and by Lemma 3.6, there is a process p_j that enters and fails in the same filter. By Lemma 3.3, p_j accesses this filter concurrently with p_i. Hence, p_j fails in some filter with location s and changes its location from s to $s+1$ in β. However, such process does not exist by the same arguments as for process p_i in *Case 2.* □

For an execution α, let α_m be the prefix with the first m events of α. The next lemma is the key to showing that the step complexity adapts to the point contention.

LEMMA 3.8. *Assume that process p_i has location s_i and that the point contention during p_i's operation is k, then for every $s' \leq s_i$, the number of processes with locations $s', \ldots, \infty$ is at most $k - s'$.*

PROOF. The proof is by induction on the length of the execution prefix. Assume that the lemma holds after m events, α_m, and that the $(m+1)$th event in α is by process p_j (p_j may be equal to p_i). We only have to consider events that change the location of p_j.

Assume this is the first operation of p_j in α; that is, p_j accesses filter 0 of chain 0. By the definition of location, the location of p_j is 1 if $turn \neq \perp$ for this filter, otherwise, its location is 0. From the definition of point contention, there are at most $k-1$ other processes accessing chains at the end of α_m. Therefore, in the case p_j has location 0, the claim holds. If $turn \neq \perp$ at the end of α_{m+1}, then chain 0 is busy at the end of α_{m+1}. Thus, there is a winner of chain 0 which is active at the end of α_{m+1} and has location 0. Therefore, there are at most $k-1$ processes with locations $1, \ldots, \infty$ at the end of α_{m+1} and the claim holds.

Assume p_j changes location from $s_j - 1$ to s_j. If $s_j > s_i$, then the claim of the lemma is not affected. Therefore, assume $s_j \leq s_i$. By the induction hypothesis, there are at most $k - s_j + 1$ processes with locations $s_j - 1, \ldots, \infty$ at the end of α_m. If at the end of α_{m+1} there is some process still with location $s_j - 1$, then at the end of α_m there are at least two processes with location $s_j - 1$ and thus, at most $k - s_j - 1$ processes with location $s_j, \ldots, \infty$. Therefore, at the end of α_{m+1} there are at most $k - s_j$ processes with location $s_j, \ldots, \infty$ and the claim holds.

If at the end of α_{m+1} there is no process with location $s_j - 1$, then by the definition of location, chain $s_j - 1$ is empty. When chain $s_j - 1$ becomes empty, the number of processes with locations $s_j - 1, \ldots, \infty$ is at most $(k - s_j + 1) - 1$. By Lemma 3.7, there are at most $k - s_j$ processes with locations $s_j - 1, \ldots, \infty$ at the end of α_{m+1}, and the claim holds. □

By this lemma, if the point contention of p_i's execution interval is k, then at most one process accesses chain $k-1$. Therefore, if p_i does not win before chain $k-1$, it accesses this chain alone and thus wins it. Hence, we have the following corollary:

COROLLARY 3.9. *If the point contention during p_i's operation is k, then p_i wins in chain $l \leq k-1$.*

According to the algorithm, if a process accesses m filters in some chain and loses, it skips the following $m-1$ chains. Therefore, the process reaches the chain it wins after accessing at most $k+1$ filters. By Lemma 3.4, it accesses $O(\log k)$ filters in the chain it wins. By the filter's code, each filter requires $O(1)$ steps. Therefore, a process executes $O(k)$ steps before it wins some chain.

LEMMA 3.10. *A process wins some chain within $O(k)$ steps.*

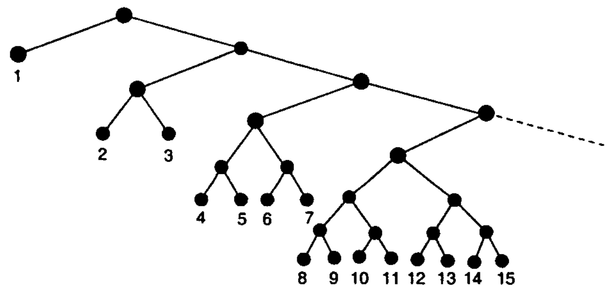

Figure 1: An adaptive tournament tree

3.2 An Adaptive Tournament Tree

After processes obtain names in the range $0, \ldots, k-1$, the process that enters the critical section is picked using an *adaptive tournament tree*. The first mutual exclusion algorithm that used a binary tournament tree is that given by Peterson and Fischer [19]. Our tree is an adaptive variant of the balanced binary tournament tree of Yang and Anderson [21]. The tree we use is an unbalanced binary tree, constructed from $\log N$ complete binary trees of exponentially growing sizes $(1, 2, 2^2, \ldots$ nodes), which are connected by a single path of nodes (Figure 1). In each inner node of the tree, a fair two-process mutual exclusion algorithm (proposed by Yang and Anderson [21]) is located. The algorithm induces $O(1)$ remote steps.

The leaves of the tree are the leaves of the complete binary trees. The leaves are numbered from left to right, so the leftmost leaf is the leaf of tree with size 1. The name obtained by a process in the renaming algorithm determines the leaf at which the process starts climbing up the tree: A process with name x_i enters the tree at the $(x_i + 1)$th leaf. A process performs the copies of the two-process mutual exclusion algorithm associated with the nodes along its path to the root, and enters the critical section by winning the root of the tree.

Since a single process starts at each leaf, only one process wins the root. The proof is similar to the tournament tree presented by Yang and Anderson [21].

Appendix A defines the tree and explains why a process with a name in the range $0, \ldots, k-1$ climbs at most $2\log k + 1$ nodes.

At each node, the execution of two-process mutual exclusion requires $O(1)$ steps. This implies that a process enters the critical section in $O(\log k)$ steps after it enters the tournament tree. The winner of each node is found in $O(1)$ time units, thus some process enters the critical section $O(\log k)$ time units after some (possibly other) process enters the tournament tree.

3.3 Complexity

Finally, we calculate the complexities of the algorithm. Let k be the point contention during an execution interval of process p_i. By Lemma 3.10, p_i is elected as a winner in some chain within $O(k)$ steps, and executes the tournament tree in $O(\log k)$ steps. Thus, the remote step complexity of the entry section is $O(k)$. When exiting from the critical section, p_i cleans all the nodes of its path in the tournament tree. The number of such nodes is $O(\log k)$. In addition, p_i updates the corresponding entry in *startFilter*. Therefore, the exit section requires $O(\log k)$ steps.

By Lemma 3.5, in $O(\log k)$ time units the winner of a busy chain is elected. In the next $O(\log k)$ time units the winner of the tournament tree is determined (according to the properties of the tournament tree). Thus, in $O(\log k)$ time units some process enters the critical section.

4. BOUNDING THE NUMBER OF FILTERS

The number of filters in a chain is bounded by recycling previously used filters. A filter can not be simply recycled, since slow processes may still be working in the filter. These processes can corrupt the filter and confuse processes that are re-using the filter. In our algorithm, the process that exits from the critical section detects "slow" processes and promotes them to enter the critical section, thus allowing filters to be recycled. Similar ideas for memory reuse appear in [1, 2, 11].

Instead of using an unbounded number of filters, a chain has only $2N$ filters, which are used cyclically. Each filter in the chain is associated with a unique process, namely, filter r is associated with process $p_{r \bmod N}$. After executing the critical section, a chain winner iterates through the filters accessed in this round, from the filter it started from in this chain to the filter it succeeded in. For each filter, it checks whether the process associated with it is still active in the current chain. If so, this process enters the critical section immediately. We say that this *slow* process is *promoted* into the critical section. After promoting a slow process, the winner stops scanning and leaves. The promoted process continues the scan from the same point upon exit from the critical section. This "takeover" mechanism frees the winner from waiting for the slow process to exit the critical section.

In addition to ensuring that recycled filters are *free* from processes, i.e., no process is executing the filter's code, the scan also initializes them. Since slow processes can corrupt initialized filters, they re-initialize each filter they could have dirtied upon leaving it.

The recycling algorithm guarantees that every time a new round in the chain starts, the next N filters from the starting filter of this round are free from processes and initialized.

The following shared variables are used by the algorithm.

- An array of chains of filters *Chains*$[0, \ldots, n-1][0, \ldots, 2N-1]$. Entry *Chains*$[l][r]$ contains filter r of chain l.
- An array *busy*$[0, \ldots, n-1]$ of Boolean variables; all entries are initially **false**. Entry *busy*$[l]$ indicates whether chain l is busy.
- An array *startFilter*$[0, \ldots, n-1]$ of integers; all entries are initially 0. Entry *startFilter*$[l]$ contains the index

of the filter in *Chains*[l] where the last round started in chain l.

- An array *endFilter*[$0, \ldots, n-1$] of integers; all entries are initially 0. Entry *endFilter*[l] contains the index of the filter in *Chains*[l] where the last round ended in chain l.

- An array *nextToClean*[$0, \ldots, n-1$] of integers; all entries are initially 0. Entry *nextToClean*[l] contains the index of the next filter in *Chains*[l] that should be cleaned (before the cleanup starts this is the first used filter in this round).

- An integer variable *nextToEnter*, initially $\perp$, contains the *id* of the process to be promoted. A process reads this variable after each step and if it is equal to its *id*, the process executes promotedEnter and enters the critical section.

- An integer variable *lastFreeChain*, initially $\perp$, contains the index of the last chain that became free. The variable is used by the promoted process to continue the cleanup protocol in this chain.

The *startFilter* array is the same as in the algorithm with unbounded memory; the *Chains* array now has $2N$ filters in each entry.

The code appears in two parts (Algorithm 2 and Algorithm 3). Since the cleanup is done by the process exiting from the critical section, we present the code of the mutual exclusion algorithm rather than the code of the renaming algorithm. Procedure getName is embedded into enter and procedure releaseName is embedded into exit.

A slow process executes promotedEnter to enter the critical section and promotedExit to exit it. The code of executeTournamentTree, cleanTournamentTree and isInChain is omitted; their semantics is clear from their names. The code of executeFilter appears in Algorithm 1.

Recall that for chain l, an execution is partitioned into *rounds*. The first round starts at the beginning of an execution; a new round starts when *busy*[l] changes from **true** to **false**. The following lemma is the key to showing the correctness of the algorithm.

LEMMA 4.1. *If a round in chain l starts at the end of α' then the filters startFilters[l], ..., (startFilters[l] + N - 1)* $\bmod 2N$ *of chain l are free from processes at the end of α'.*

SKETCH OF PROOF. The proof is trivial when the second part of the array is not used at the end of α'. Let *start* be the value of *startFilters*[l] at the end of α'. We consider some process p_i and show that p_i is not accessing filters $start, \ldots, (start + N - 1) \bmod 2N$ at the end of α'.

By the algorithm, filters $(start - N) \bmod 2N, \ldots, (start - 1) \bmod 2N$ were in use after the filters $start, \ldots, (start+N-1) \bmod 2N$ were in use last time before α'. Between these filters there must be a filter associated with p_i. Thus, there is a prefix α'' of α' such that the entry associated with p_i was cleaned at the end of α'' and the filters $start, \ldots, (start + N - 1) \bmod 2N$ are not used in the interval between α'' and α'. The cleanup protocol ensures that either p_i was not in chain at the end of α'', or it was promoted to enter the critical section in α''. Thus there is a point in the interval between α'' and α when p_i is not in the chain.

Hence, if process p_i is accessing chain l, it began to do this after α''. By the algorithm, filter $(start - 1) \bmod 2N$ is the filter where the winner has succeeded in the previous round in chain l. A process can enter filter r of the chain only by succeeding in $(r-1) \bmod 2N$. In addition, a process cannot pass the filter where the winner of the round has succeeded in. Therefore, at the end of α', p_i has not passed filter *start* and is not accessing the filters $start, \ldots, (start + N - 1) \bmod 2N$ at the end of α'. Thus, filters *startFilters*[l], ..., (*startFilters*[l] + N - 1) $\bmod 2N$ are free from processes at the end of α'. □

Algorithm 2 Adaptive mutual exclusion with bounded memory (part 1)

```
private variables
    l : integer

procedure enter()
l := 0
index := −1
repeat
    l := l + index + 1
    if (! busy[l]) then
        busy[l] := true
        ⟨result, index⟩ := executeChain(l)
    else index := 0
until result = win
executeTournamentTree(l)
            // execute the tournament tree from l'th leaf

procedure exit()
nextToClean[l] := startFilter[l]
if cleanUsedFiltersInChain(l) then
    cleanTournamentTree(l)
    startFilter[l] := (endFilter[l] + 1) mod2N
    busy[l] := false

procedure executeChain(l)
Filters := Chains[l]
curr := startFilter[l]
while (true)
    if executeFilter(Filters[curr]) = win then
        if curr ≠ startFilter[l]
                and ¬Filters[curr − 1].c then
            endFilter[l] := curr
            return ⟨win, (curr − startFilter[l]) mod 2N⟩
        else(curr + +) mod 2N
    else return ⟨lose, (curr − startFilter[l]) mod 2N⟩
endwhile
```

A slow process can determine whether the round in the chain it is accessing has changed by checking if the value of *startFilters*[l] changed since it began chain l. It performs

Algorithm 3 Adaptive mutual exclusion with bounded memory (part 2)

```
procedure promotedEnter(filter)
                    // entry section for promoted process
  cleanFilter(filter)

procedure promotedExit()
                    // exit section for promoted process
  nextToEnter =⊥
  l := lastFreeChain
  if cleanUsedFiltersInChain (l) then
      cleanTournamentTree(l)
      startFilter[l] := (endFilter[l] + 1) mod2N
      busy[l] := false

procedure cleanUsedFiltersInChain(l)      // clean used filter
  repeat
      if nextToClean[l] ≠ id
            and isInChain(nextToClean[l]) then
          lastFreeChain := l
          nextToEnter := nextToClean[l]
          return false
      cleanFilter( Chains[l][nextToClean[l]])
      nextToClean[l] := (nextToClean[l] + 1) mod2N)
  until (nextToClean[l] = endFilter[l])
  return true

procedure cleanFilter(filter)          // clean a specific filter
  filter.b = false
  filter.d = false
  filter.c = false
  filter.turn = ⊥
```

this check each time before moving to the next filter in the chain. If the value does not change, then this is still the same round in the chain.

Cleanup guarantees that before *startFilters*[l] gets the same value again, the entry associated with the process was cleaned, and if the process was still in the chain, it was promoted to enter the critical section.

Procedure cleanUsedFiltersChain preserves mutual exclusion, because the chain is busy as long as the last filter of this round is not cleaned yet. Therefore, it is not possible that one process enters the critical section via promotedEnter, and the other via regular enter. The cleanup stage is the only change to Algorithm 1, and by Lemma 4.1 there are enough free filters at the beginning of each round. Therefore, Algorithm 2 guarantees mutual exclusion.

The remote step complexity of cleanUsedFiltersChain is $O(\log k)$ since a process cleans $O(\log k)$ filters and each filter is cleaned $O(1)$ steps. The space complexity of the algorithm is dominated by the size of the array *Chains*, which is $O(nN)$.

5. REDUCING THE SPACE COMPLEXITY

The space complexity of our algorithm is a function of N, the range of process names; N may be very large compared to the total number of processes, n. Thus, transforming the space complexity of the algorithm to be a function of n, could improve it significantly. This is done by having each process execute one-shot n-renaming before its first entry to the critical section, and using the obtained name from there on. Once a process receives a name, it never releases this name.

Algorithm 4 Procedures for one-shot renaming.

```
procedure getSmallName()
  ch := entryChain
  index := −1
  result := lose
  repeat
      ch := ch + index + 1
      if (ch < entryChain)
          ch := entryChain
      ⟨result, index⟩ := executeChain(NamesChains[ch])
  until result = win
  name := ch

procedure updateEntryChain ()
  if (entryChain ≤ name)
        // move the next chain after name, if it was before
      entryChain := name + 1
```

The step complexity of known one-shot renaming algorithms depends on the number of allocated names. Since processes that exit the critical section do not release their names, employing one of these algorithms will cause the step complexity of the resulting mutual exclusion algorithm to depend on n and not on k.

The solution employs the fact that one-shot renaming is used as part of a mutual exclusion algorithm. We introduce a one-shot non-wait-free n-renaming algorithm, with $O(\log k)$ system response time and $O(k')$ remote step complexity, where k' is the *interval* contention.

This algorithm uses the same array of chains of filters as long-lived renaming algorithm described in Section 3. Unlike the long-lived renaming algorithm, the entry point to the array, which is initially the first chain, is not fixed. Each process that leaves the critical section makes sure that the entry point to the next chain is after its name.

The data structures that serve the one-shot renaming algorithm (which are distinct from those used by the mutual exclusion algorithm itself) are as follows. The array of chains is the *NamesChains* array. A new shared integer variable *entryChain* contains the index of the chain from which the next call to getSmallName starts to execute.

Algorithm 4 contains the code of procedures getSmallName and updateEntryChain. Procedure enter should be modified to check whether the process has a small name and call getSmallName if it does not. Hence, getSmallName is called only when the process executes enter for the first time. After that, the name of the process remains the same throughout the algorithm. Procedure updateEntryChain is called by each process upon exit from the critical section.

The properties of chains that have been proved for our first algorithm guarantee that no two processes receive the same name, and that the range of names is $0, \ldots, n-1$.

A process that starts getSmallName skips the names occupied by inactive processes. This makes the remote step complexity of getSmallName be $O(k')$, where k' is the interval contention. Thus, $O(k')$ is the remote step complexity of the first call to enter. The remote step complexity of all subsequent calls to enter does not change. Since getSmallName is called only once, the amortized remote step complexity of the algorithm is not affected.

The mechanism for updating and checking the entry chain turns the system response time of getSmallName to be identical to the system response time of the algorithm from Section 3. Therefore, the system response time of the whole algorithm remains $O(\log k)$, where k is a point contention.

The space complexity of the one-shot renaming algorithm is $O(n \log n)$, since n chains are used and each chain contains $\log n$ filters. The space complexity of our mutual exclusion algorithm is n times the range of names, that is, $O(n^2)$.

6. DISCUSSION

We presented a mutual exclusion algorithm which adapts to point contention; this is the strongest notion of adaptiveness known in the literature. The algorithm has $O(k)$ remote step complexity and $O(\log k)$ system response time. We showed how to make the space complexity of the algorithm depend only on n.

Cypher [12] has shown that there is no mutual exclusion algorithm with constant remote step complexity. It is an obvious open problem to improve the remote step complexity of our algorithm or to show that it is optimal.

Anderson and Kim [7] mention an algorithm with $O(k)$ remote memory references, however, the system response time of this algorithm is $O(k)$. Since two process mutual exclusion of Yang and Anderson is used in the nodes of our adaptive tournament tree, it has $O(\log k)$ remote memory references. Thus, designing a long-lived renaming algorithm with adaptive number of remote memory references and system response time of $O(\log k)$ will result in a mutual exclusion algorithm with the same properties.

7. REFERENCES

[1] Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Adaptive long-lived renaming using bounded memory. Unpublished manuscript available at www.cs.technion.ac.il/~hagit/pubs/AAFST99disc.ps.gz, Apr. 1999.

[2] Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-lived renaming made adaptive. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 91–103, 1999.

[3] Y. Afek, G. Stupp, and D. Touitou. Long-lived adaptive splitter and applications. Unpublished manuscript, Dec. 1999.

[4] Y. Afek, G. Stupp, and D. Touitou. Long-lived and adaptive collect with applications. In *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science*, pages 262–272, Oct. 1999.

[5] R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Proceedings of the Real-Time Systems Symposium*, pages 12–22, Dec. 1992.

[6] J. Anderson and Y. J. Kim. Fast and scalable mutual exclusion. In *Proceedings of the 13th International Symposium on Distributed Algorithms*, pages 180–194. Springer-Verlag, Sept. 1999.

[7] J. Anderson and Y.-J. Kim. A new fast-path mechanism for mutual exlusion. Unpublished manuscript, 1999.

[8] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–548, July 1990.

[9] H. Attiya and A. Fouren. Adaptive wait-free algorithms for lattice agreement and renaming. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 277–286, 1998. Full version available at www.cs.technion.ac.il/~hagit/pubs/AF98rev.ps.gz.

[10] J. E. Burns and G. L. Peterson. The ambiguity of choosing. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 145–158, 1989.

[11] M. Choy and A. K. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing*, 8(1):1–17, 1994.

[12] R. Cypher. The communication requirements of mutual exclusion. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 147–156, 1995.

[13] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[14] M. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[15] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 18(8):453–455, Aug. 1974.

[16] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, Feb. 1987.

[17] M. Merritt and G. Taubenfeld. Speeding Lamport's fast mutual exclusion algorithm. *Inf. Process. Lett.*, 45, 1993.

[18] M. Moir and J. H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Programming*, 25(1):1–39, Oct. 1995.

[19] G. L. Peterson and M. J. Fischer. Economical solutions for the critical section problem in a distributed system. In *Proceedings of the 9th ACM Symposium on Theory of Computing*, pages 91–97, 1977.

[20] E. Styer. Improving fast mutual exclusion. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 159–168, August 1992.

[21] J.-H. Yang and J. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, Aug. 1995.

APPENDIX

A. AN ADAPTIVE TOURNAMENT TREE

Our adaptive tournament tree is similar to the adaptive tree used for lattice agreement [9]. This is an unbalanced binary tree T_r defined inductively as follows: T_0 consists of a root v_0 with a single left child. For $r \geq 0$, suppose T_r is defined with an identified node v_r, which is the last node in the in-order traversal of T_r; notice that v_r does not have a right child in T_r. T_{r+1} is obtained by inserting a new node v_{r+1} as the right child of v_r, and inserting a complete binary tree C_{r+1} of height $r+1$ as the left subtree of v_{r+1}. By the construction, v_{r+1} is the last node in an in-order traversal of T_{r+1}.

By the construction, the leaves of T_r are the leaves of the complete binary subtrees $C_0, C_1, \ldots, C_r$. Therefore, the total number of leaves in T_r is $\sum_{j=0}^{r} 2^j = 2^{r+1} - 1$. The proof of the next lemma appears in [9].

LEMMA A.1. *Let w be the i-th leaf of T_r, $1 \leq i \leq 2^{r+1}-1$, counting from left to right. Then the depth of w is* $2\lfloor \log i \rfloor + 1$.

We use $T_{\log n}$, which has n leaves (for simplicity, we assume that n is a power of 2).

A process with a new name x_i starts the algorithm at the $(x_i + 1)$th leaf of the tree, counting from left to right. Since $k \leq n$, $T_{\log n}$ has enough leaves for names in a range of size k.

By Lemma A.1, a process starts in a leaf of depth at most $2\lfloor \log k \rfloor + 1$. Therefore, p_i accesses at most $2 \log k + 1$ nodes.