# Mutual Exclusion with $O(\log^2 \log n)$ Amortized Work

Michael A. Bender
Stony Brook University & Tokutek, Inc.
bender@cs.stonybrook.edu

Seth Gilbert
National University of Singapore
seth.gilbert@comp.nus.edu.sg

**Abstract**— This paper presents a new algorithm for *mutual exclusion* in which each passage through the critical section costs amortized $O(\log^2 \log n)$ RMRs with high probability. The algorithm operates in a standard asynchronous, *local spinning*, shared-memory model with an oblivious adversary. It guarantees that every process enters the critical section with high probability. The algorithm achieves its efficient performance by exploiting a connection between mutual exclusion and approximate counting.

## 1. Introduction

Coordinating access to shared resources is a fundamental problem in parallel computing. In the classic problem of *mutual exclusion*, introduced by Dijkstra [10], each process attempts to gain exclusive access to some shared resource. Whenever a process gains exclusive access, it can safely execute its *critical section*.

There have been hundreds of papers written on mutual exclusion; see, e.g., [1]–[4], [8], [11], [14]–[16], [19]. Performance of a mutual-exclusion algorithm is typically measured in terms of *remote memory references* or *RMRs*. The assumption is that each process has a local memory/cache, which it can access cheaply, and a read/write shared memory, which is expensive to access—these are the RMRs. Processes thus have the capacity to perform *local spinning* for free, i.e., to spin-wait on a local variable until it changes. (Without local spinning, efficient mutual exclusion is impossible [19].)

Until recently, the most efficient mutual-exclusion algorithms, such as the one by Yang and Anderson [19], used $O(\log n)$ RMRs per passage on a system of $n$ processes.

It was recently proved that this bound is optimal for deterministic algorithms [8], [11]. In 2009 Hendler and Woelfel [14] showed that randomized algorithms can perform better than deterministic algorithms by demonstrating one that achieves $O(\log n/\log\log n)$ expected RMRs per passage.

Most efficient prior solutions (typified by [14], [19]) are based on a *tournament tree* construction. A process's passage begins at the leaf of a tree. Processes compete to climb the tree. When a process reaches the root, it executes its critical section and then exits the tree, allowing other processes to continue competing. Since the degree of the tree must be relatively small (to ensure that competition at each node is efficient), it is hard to achieve performance that is much better than $O(\log n)$ RMRs per process.

*Results:* This paper represents a departure from previous work in terms of techniques, adversary model, and performance. We give a mutual exclusion protocol where each process performs only $O(\log^2 \log n)$ amortized RMRs per passage with high probability. Our protocol is randomized and works against an oblivious adversary; every process enters the critical section with high probability.

*Mutual Exclusion and Counting:* Our work exploits a connection between mutual exclusion and approximate counting. It should not be surprising that such a connection exists. Consider, for example, a standard "mutual exclusion" mechanism found in retail stores everywhere: each customer takes a number, issued in sequential order, and the customers are serviced in the order of their numbers. (Lamport famously exploited this idea in the *Bakery Algorithm* [17].)

There are several difficulties in putting this connection to good use. First, mutual-exclusion algorithms have generally possessed better complexities than counting algorithms. Prior to [5], the best concurrent-counter protocols took $\Omega(n)$ steps per increment, and even the best *approximate* counting protocols were just barely sublinear [6]. Thus, it might have seemed unlikely that reducing mutual exclusion to counting would yield asymptotically good results. It was an exciting development when Aspnes et al. [5] gave an elegant wait-free data structure for exact counting with only $O(\log^2 n)$ steps per increment and $O(\log n)$ steps per read. These counting bounds are still exponentially larger than our goal of $O(\log^2 \log n)$ RMRs for mutual exclusion. However, we can leverage the counter construction of [5] to build an *approximate* counter in which increment operations take $O(\log^2 \log n)$ steps.

A further difficulty is that wait-free counter (and approximate-counter) constructions only support increment, not decrement, operations. Ideally, a process would increment a counter on beginning its passage and decrement a counter on completing its passage. In this case, the value of the counter would indicate the number of active processes. This information would significantly simplify the construction of a mutual exclusion algorithm. Unfortunately, it appears difficult to support both increment and decrement efficiently (even for exact counters).

This research was supported in part by NSF Grants CCF 0937822, CCF 1114809, CCF 0634793, and CCF 0540897, DOE Grant DE-FG02-08ER25853, and NUS FRC R-252-000-443-133.

This paper takes advantage of the connection between (approximate) counting and mutual exclusion by weakening what we actually require from the counter, i.e., by relying on an approximate counter with no decrement operation.

*Adversary Models:* While we give exponentially better bounds than previous papers (e.g., [15], [16], [19]), we cannot claim that our results subsume these earlier papers, because we depart from the adaptive-adversary model, instead assuming an oblivious adversary. It has been conjectured [18] that for an adaptive adversary, $\Omega(\log n / \log\log n)$ RMRs per process is a lower bound (for starvation-free algorithms). This would imply that a weaker adversary is essential to obtain our improved bounds. This paper is the first to beat the $O(\log n / \log\log n)$ RMR bound for some adversary.

An oblivious adversary models many but not all sources of asynchrony. Specifically, it models asynchrony whose sources are independent of the choices made by the mutual-exclusion protocol (e.g., speed changes caused by other programs competing for CPU cycles or memory bandwidth).

It might be interesting to execute our algorithm in parallel with an existing algorithm [15], [16], [19] that can tolerate an adaptive adversary (using a simple lock to mediate winners of the two protocols). The resulting combination would be fast when the adversary is oblivious, while still guaranteeing good results when the adversary is not.

*Model:* We consider $n$ processes, each of which accesses the critical section at most once. (This simplifies the presentation, because we can associate each passage with a unique process. Generalizing to a polynomial number of accesses per process follows trivially.) Each process takes an arbitrary (unbounded) number of steps, and the *oblivious adversary* (acting as a scheduler) decides, for each execution, the order in which processes take steps. An oblivious adversary makes these choices with full knowledge of the initial state, including the mutual-exclusion protocol, but without knowing the outcomes of random coin flips of the processes. Thus, an oblivious adversary scheduler determines the entire schedule prior to the execution.

Since modern architectures cache memory aggressively, we model memory accesses as per the *cache-coherent* (CC) model. Each process keeps a local cache of some variables from the shared memory. Whenever the value of a variable is cached locally, a process can read it for free. Whenever a variable is written, all caches except for that of the writing process are invalidated; the next read by each process (except for the writing process) costs a normal shared memory access, i.e., an RMR.

We make use of local (cached) memory for the purpose of spinning. While executing the protocol, processes may *register* events, meaning they indicate a location in memory to monitor, and a *callback* function to execute if that location in memory is modified. During registration, this memory is read into cache, at which point it is continually monitored locally. If that location in memory ever changes, the cache is invalidated, and the process discovers that the memory has changed. This wakes/interrupts the process, and the previously registered callback function is executed. Notice that registering an event and discovering that it has been triggered each cost one RMR, while the remaining local monitoring is free. All local spinning will be captured via this mechanism; every other read or write operation is assumed to cost an RMR.

We also assume, without loss of generality, that processes can execute *compare-and-swap* (CAS) operations. We can transform each CAS operation into a set of read and write operations using the construction of Golab et al. [12]. They show that each CAS operation requires $O(1)$ RMRs.[1] A $\mathsf{CAS}(v, old, new)$ operation compares $v$ and $old$; if they are equal, then it atomically sets $v$ to $new$.

## 2. Algorithm Overview

We now give an overview of the mutual exclusion algorithm. We begin with high-level ideas. We progressively summarize problems and their solutions until the full algorithm comes into focus.

### 2.1. High-level Ideas

Assume that we have a counter $C$ that supports both increment and decrement operations. In this case, every process increments $C$ when it begins its passage and decrements $C$ when it completes its passage. At any given time, reading $C$ yields a count of the number of active processes.

After incrementing counter $C$, a process reads the counter and uses the value to find a free spot in the *waiting array* $A$, which is used for local spinning while processes wait their turn to enter the critical section. Specifically, if the counter $C$ returns value $k$, then the process randomly searches for a spot in the first $\Theta(k)$ slots in the array. Since the counter $C$ accurately estimates the number of active processes, at most a constant fraction of the first $\Theta(k)$ slots are *full*; hence each random probe has a constant probability of finding an empty slot. Once the process finds a spot, it goes to sleep, spinning until it is awakened.

When a process exits the critical section, it searches for a replacement process in the array $A$, handing off control of the critical section. It proceeds by reading the counter $C$—assume that $k$ is the value returned by $C$—and searching randomly for a process in the first $\Theta(k)$ slots of the array. If there are $k$ active processes waiting in the first $\Theta(k)$ slots of the array, each random probe has a constant probability of finding an occupied slot. At this point, the exiting process removes itself from the array $A$, wakes the process occupying the selected slot, and decrements the counter $C$.

[1] Their implementation is *strongly linearizable*, and hence can be safely used in the presence of an oblivious adversary; see [13].

There are problems with this basic protocol. First, we have to address how the *very first* process enters the critical section. Second, we have to cope with the case when many processes have incremented the counter, but not yet joined the array. Third, we can only use counters that increment, not decrement.

In Section 2.2 we provide a base protocol that addresses these issues. The resulting protocol is inefficient, but illustrates the basic ideas of our protocol. In Section 2.3, we show how to replace the exact counters used in Section 2.2 with approximate counters.

Finally, in Section 2.4, we address the problem of decrementing counters as follows. We maintain two separate counts: the (approximate) number of increments and the (exact) number of decrements. As long as the number of increments is sufficiently larger than the number of decrements, then we can ignore the decrements and the counter still yields a constant-factor approximation. When the number of decrements becomes sufficiently large, we reset the counters, beginning a new epoch.

This resetting, however, creates a *chicken-and-egg problem*: the first step that a process takes in an epoch must be a write—a process must *make a mark*—specifically, one that is visible to other processes. If the first step is a read, then a set of slow processes could wake up, take invisible steps, sleep until the epoch finishes, wake up in the next epoch, take invisible steps, sleep until the epoch finishes, and so on, without accomplishing useful or visible work. In this case, the total work could grow too large because of the invisible read operations. On the other hand, if the first step of a process in an epoch is a write, how does the process discover the epoch number (i.e., where to find the data structure for the given epoch)? We address these issues in Section 2.4.

### *2.2. Base Mutual Exclusion Protocol*

The basic mutual exclusion protocol consists of four components: (i) a variable $lock$ to guard the critical section; (ii) an array $A$, which processes use while waiting for the critical section to become free; (iii) a counter $C$ that processes increment immediately on joining the system, and (iv) a counter $join\text{-}count$ that processes increment after they have joined the array $A$ (but before any spinning occurs).

The lock is implemented via a CAS operation: when a process $p$ wants to claim the lock, it executes $\mathsf{CAS}(lock, 0, p)$. If the CAS operation succeeds, then $p$ has acquired the lock. When a process $p$ wants to release the lock, it simply writes $0$ to $lock$. The counters $C$ and $join\text{-}count$ can be implemented using the construction in [5].

A process $p$ enters the critical section as follows. First, it increments the counter $C$. It then repeats the following until it succeeds in claiming a slot in $A$: (i) it reads value $k \leftarrow C.\mathsf{read}()$; (ii) it chooses a random location $\ell$ in array $A$ in the range $1\,..\,\Theta(k)$; and (iii) it attempts to claim slot $\ell$ by executing $\mathsf{CAS}(A[\ell], 0, p)$. When the CAS succeeds, process $p$ increments counter $join\text{-}count$. It then tries to acquire the $lock$. If the process succeeds, it enters the critical section. Otherwise, it spins on the array slot $A[\ell]$.

A process $p$ exits the critical section as follows. First, $p$ sets the array slot $A[\ell] \leftarrow -1$, indicating that the slot is now empty. Next, $p$ releases the lock. Finally, $p$ repeats the following steps: (i) it reads $c_1 \leftarrow C.\mathsf{read}()$; (ii) it reads $c_2 \leftarrow join\text{-}count.\mathsf{read}()$; (iii) if $c_1 > c_2$, then $p$ exits; otherwise, (iv) it chooses a random location $\ell$ in the range $1\,..\,\Theta(c_1)$; and (v) if $A[\ell] > 0$, then it signals to the process spinning at $A[\ell]$ to wake up and exits. Otherwise, $p$ repeats (1)–(v).

This protocol ensures mutual exclusion, as no process enters the critical section without acquiring the lock. The protocol also guarantees liveness: Assume process $p$ spins forever. Since $p$ fails to acquire the lock, we know that at least one process succeeds in entering the critical section. Let $q$ be the last process to exit the critical section. Observe that $q$ necessarily releases the lock at some point after $p$ tries to acquire it. Since $q$ is the last process to exit the critical section, it does not find (and awaken) another process in $A$. Thus, we conclude that $q$ exits on finding $c_1 > c_2$. This exit condition implies that some process $p'$ has incremented $c_1$ but not $c_2$; $p'$ will proceed to try to acquire the lock after $q$ has released it, ensuring that either $p'$ or some other process enters the critical section after $q$, which is a contradiction.

Finally, we observe that the algorithm is reasonably efficient (i.e., on average each process performs $O(1)$ counter increments and other operations) *as long as* the number of processes that have exited the critical section is at most a constant fraction of the total number of processes that have incremented counter $C$. To see this, there are two parts of the protocol that we have to examine.

First, looking for a free slot in the array $A$: since a process reads the counter $C$ prior to searching for a spot in the array, either it succeeds with constant probability, or the number of processes in the system has doubled since it read $C$. In the latter case, we can amortize the cost of the read against the newly arrived processes.

Second, finding a process in the array $A$, after exiting the critical section: we search for a process in $A$ only when $c_1 = c_2$, i.e., all the processes that have begun joining have found a slot in the array $A$. Thus, at least a constant fraction of the slots in the range from $1\,..\,\Theta(c_1)$ are either marked by processes that are spinning, or by processes that have already exited the critical section. If the number of processes that have completed is at most a constant faction of the number of processes that have begun joining, then each probe in $A$ has a constant probability of finding a spinning process.

On the other hand, if most of the processes have already completed the critical section (and the array $A$ is empty), then this protocol becomes inefficient.

### 2.3. Approximate Counters

We now observe that we can replace the exact counters with approximate counters. In Section 4, we show how to implement approximate counters where each increment and read operation requires at most $O(\log^2 \log n)$ steps. Although the exact counter works even with a strong adversary, our approximate counter requires a weaker adversary.

Replacing the exact counter with the approximate counter creates two potential problems. First, we use the value $k$ read from the counter $C$ to find a spot in the array $A$. Since the counter is within a constant factor of the correct value, it remains easy to see that the range $1\,..\,\Theta(k)$ is big enough to contain all the processes that have joined.

The second problem arises when a process exits the critical section, since the values $c_1$ and $c_2$ may each be off by a constant factor. Instead of exiting when $c_1 > c_2$, a process exits if for some constant $0 < \varepsilon < 1$ (which depends on the approximation factor of the counters), the value $c_1 > c_2/\varepsilon$. In this case, at least one process has incremented $C$ and not $join\text{-}count$, and hence it is safe for a process leaving the critical section to exit. Otherwise, if $c_1 \leq c_2/\varepsilon$, we know that some $\Theta(c_1)$ processes have completed joining and hence we can find one in the array $A$ in the range $1\,..\,\Theta(c_1)$. Thus, using approximate counters instead of exact counter, the mutual exclusion protocol maintains the properties previously discussed.

### 2.4. Resetting the Counters

Finally, we address the issue of efficiency. We keep a count $LCount$ of the number of processes that have completed the critical section. When $lCount$ reaches a constant fraction of $C$, we reset the data structure and begin a new *epoch*. We cannot, however, simply create a new copy of the data structure (i.e., counters $C, join\text{-}count$, array $A$) for the new epoch and move all the processes to the new data structure. The (chicken-and-egg) problem is as follows.

Assume that in every epoch at least $T = \Theta(\log^4 n)$ processes complete (for reasons to be explained later). Thus, there may be $\Theta(n/T)$ epochs. If each epoch has its own data structure (i.e., counters and arrays), then the first step a process takes in an epoch must be to read some epoch counter that specifies which data structure to use. Consider some $\Theta(n)$ processes that read the epoch counter without modifying the data structure in any other way, and then stall until the epoch ends. Since the large batch of stalled processes is invisible, each epoch must come to an end after some $T$ processes join, enter the critical section, and leave. The total number of steps taken by the batch of stalling processes, after $\Theta(n/T)$ epochs, is $\Theta(n^2/T)$, which is too much.

Instead, we insist that in its first step in an epoch, a process *makes a mark*, i.e., begins by incrementing counter $C$. We instantiate three copies of counter $C$ which we call $C[0]$, $C[1]$ and $C[2]$. When a process increments counter $C$, it simply chooses one of the three counters at random to increment. In epoch $e$, we read from counter $C[e \mod 3]$. With constant probability, a process performing an increment chooses the correct counter for the current epoch. Thus, the value read from counter $C[e \mod 3]$ remains a constant-approximation of the correct count. (Notice that a further problem is ensuring that one write step by a process is sufficient to ensure that it is counted, as incrementing the counter may take more than one step; this is achieved by a *helping mechanism* and is explained further in Section 5.)

Each epoch maintains its own specific copy of counter $join\text{-}counter$ and array $A$. After process $p$ increments the randomly selected counter $C[\cdot]$, it reads the epoch counter and proceeds to use the correct instance of the data structure.

Whenever an epoch ends, the counter $C[(e-1) \mod 3]$ is cleared, and the epoch counter is incremented. At the same time, all the processes spinning in the array $A$ for the old epoch are awakened, and repeat the entire protocol in the new epoch (i.e., incrementing $C[\cdot]$, joining $A$, incrementing $join\text{-}counter$). Since the counter $C[\cdot]$ is a good approximation of the total number of processes that have taken *even one step*, and we use the value of counter $C[\cdot]$ to determine when to end the epoch, we can amortize the work done by processes moving from one epoch to the next against the work done by processes that complete in that epoch.

The remaining problem is that, since we choose a counter $C_1$ at random, the estimate is good only after a polylogarithmic number of processes join. (The estimate can be too low; it can never be more than a factor of 3 too big.) Thus, we only rely on the counter $C[\cdot]$ when at least $T = \Theta(\log^4 n)$ processes have joined it. To cope with this problem, we will in parallel execute a second (deterministic) mutual exclusion instance which only $T$ processes are allowed to use. We can construct this secondary instance using existing techniques where each process performs at most $O(\log T)$ steps.

There are also three places we rely on counter $C[\cdot]$. First, when we are searching for a spot in the array $A$, if the value read from the counter is smaller than $T$, we round the value up to $T$. Second, when we are searching for a process in the array $A$, if the value read from the counter is smaller than $T$, we round the value up to $T$. Finally, when comparing $c_1$ and $c_2$ (to determine if a process exiting the critical section can safely depart), we simply exit if the counter $c_1 < \Theta(T)$: in that case, it is safe to assume that the *small mutex* instance will send a process into the critical section.

## 3. Background

This section describes four basic building blocks.

### 3.1. Max-Registers

A max-register is an object that stores the largest value ever written to it. A max-register supports two operations: read and write, where the read operations returns the largest value ever written. A max-register is parameterized by

a value $v_{\mathsf{max}}$ that specifies the maximum allowed value. Aspnes et al. [5] describe a max-register in which each operation has cost $O(\log v_{\mathsf{max}})$.

### 3.2. Exact Activity Counter

An activity counter has a set of ports $P$, and supports two operations: (i) join$(p)$, for some port $p \in P$, and (ii) read$()$. The read operation returns a count of the number of ports for which there has been at least one join operation. (If there are two join operations executed on the same port, then the count is only incremented by one.) The counter construction in [5] immediately yields an exact activity counter with cost $O(\log |P|)$ to read and $O(\log^2 |P|)$ to increment.

### 3.3. Small Bounded Counter

We also need a traditional counter that returns the exact number of increments. The maximum value of the counter will be bounded by some small value $v_{\mathsf{max}}$. The bounded-counter construction uses an exact activity counter of size $2v_{\mathsf{max}}$. To increment the counter, a process repeatedly chooses and joins a port $p \in \{1, \ldots, 2v_{\mathsf{max}}\}$ at random until it finds one that is free, or it discovers that the counter has exceeded $v_{\mathsf{max}}$. Notice that each attempt to find a port succeeds with probability at least $1/2$.

One additional feature of this counter is that it returns the port number claimed during the increment, if any. Thus, it will return a valid port identifier to at least $v_{max}$ and at most $2v_{max}$ processes.

*Lemma 1:* Assume $\log n \leq v_{\mathsf{max}} \leq n$. For every $c$, for a set of $k$ join operations, collectively they cost $O((k + v_{\mathsf{max}}) \log^2 v_{\mathsf{max}})$ RMRs, with probability $\geq 1 - 1/n^c$. □

Observe the distinct semantics between these two counters: the bounded counter does not require the calling process to provide a port number, and counts every increment; the exact activity counter assumes the caller has a port number, and counts the number of ports that have been joined.

### 3.4. Deterministic Mutual Exclusion

When there are only a small (e.g., polylogarithmic) number of active processes, we rely on a deterministic mutual exclusion protocol (e.g., [19]). The protocol has $P$ ports, i.e., can be accessed by up to $|P|$ different processes, where $|P|$ will be polylogarithmic. It supports two operations: (i) join$(p,$ on-win-mutex$)$, which accesses the mutual exclusion object on port $p$ and executes the function on-win-mutex when the critical section is obtained; and (ii) leave$(p)$, which indicates that the process on port $p$ is leaving. The protocol consists of a tree with $|P|$ leaves where each node has a lock that can be claimed (and released) via a CAS operation. On joining port $p$, a process claims the lock at the leaf. It then attempts to walk up the tree as in [19], claiming each lock on the leaf-to-root path. If a lock is already taken, the process spins locally, waiting for it to become free. When a process leaves (or completes the critical section), it walks down the tree, releasing the locks.

## 4. Approximate Activity Counter

This section describes an approximate activity counter, approx-counter, that counts the number of processes that have executed a join operation. It also supports a clear that resets the counter to zero; however, we require that the clear operation is never executed by more than one process concurrently. With high probability, the read operation returns a constant-fraction approximation of the number of processes that have joined the counter since it was last cleared.

*Basic Idea:* Here we adapt a standard trick: if $k \leq n$ independent variables $\langle X_1, \ldots, X_k \rangle$ are each exponentially distributed on the range $[1, \ldots, \log n]$, then in expectation, $\max(X_j) = \log k$ (to a suitable approximation). If each joining process chooses a value at random according to the exponential distribution and writes that value to a max-register, then the max-register stores an estimate of the number of processes that have joined.

To achieve a high-probability estimate of the number of joins, we modify this scheme slightly: a joining process only writes a value $j$ to the max-register if at least $\log n$ other processes have also randomly selected $j$. For each of the $\log n$ possible values of $j$, we use an activity counter to determine whether sufficiently many processes have chosen $j$.

Observe that the approximate counter does not work if the adversary is adaptive, because an adaptive adversary could bias the estimate by delaying the small number of processes that randomly select large values.

*Detailed Description:* The counter is parameterized by a constant $c$. The data structure consists of four parts: (i) a max-register $M$, (ii) an array of $\log n$ exact activity counters, each with $c \log^2 n$ ports, (iii) a two-dimensional array of bits $L$ that contains one bit for every port on every counter, and (iv) a bounded counter. To clear the counter, we simply allocate new memory for the max-register, exact activity counters, and bounded counter, thus allowing each to be "atomically" cleared. (Thus we can view $M$ and $C$ and $sC$ as "pointers" to the specified data structures.') The array $L$ is cleared sequentially (so that there is no need to read a pointer to find the location of array $L$).

We define a threshold $T = \Theta(\log^4 n)$ differentiating "small activity" and "normal activity": when there are fewer than $T$ active processes, we rely on the bounded counter; when there are more than $T$ active processes, we rely on the estimate derived from the max-register $M$ and the exact activity counters. This is necessary since the estimate derived in this fashion is only accurate when a polylogarithmic number of processes have joined.

Joining the approximate counter consists of two parts: a pre-join and a join. In the pre-join a process chooses an exact activity counter at random using an exponential distribution, and it chooses a port on that counter using a uniform distribution. It then marks the bit in array $L$ associated with this counter and port. By making this mark

immediately during the pre-join, slower processes can have their join finished by faster processes. After the pre-join, a process calls join specifying the previously chosen counter and port. The process then checks whether the bit in the array $L$ remains set (i.e., there has been no intervening clear), and if so, the process joins the specified exact activity counter at the specified port. If the counter value is sufficiently big, then the process writes the counter id to the max-register. Finally, the process, increments the bounded counter.

A read operation examines both the bounded counter and the approximate counter. If the bounded counter exceeds the threshold $T$, then it returns the maximum of $T$ and $2^m \log n$, where $m$ is the value read from the max-register. Otherwise, it simply returns the value of the bounded counter.

One risk is that a slow process may never get counted. The array $L$ is the location where a process first "makes its mark." The help procedure is executed by faster processes, ensuring that every process that makes a mark is counted. To help, a process chooses a random entry in the array $L$, and if there is a mark, then it completes the join for the process that made a mark there. After $\Theta(\log^4 n)$ helping operations, with high probability, every process that has made a mark has been counted.

*Analysis:* We now argue that the counter returns a constant factor approximation.

*Lemma 2:* For every constant $c$, there exists a constant $0 < \delta < 1$ (as a function of $c$) such that: Let $z$ be the value returned by some read() operation $\tau$. Assume that there are no clear operations concurrent with $\tau$. Let $\kappa$ be the most recent clear operation that completed (if any exists) prior to the beginning of $\tau$. If there exists a set of $p$ processes that begin executing the join procedure after clear operation $\kappa$, and completed the join procedure prior to the read operation $\tau$, then $z \geq \delta p$ with probability at least $1/n^c$. □

*Lemma 3:* For every constant $c \geq 3$, there exists a constant $\gamma > 1$ (as a function of $c$) such that: Let $z$ be the value returned by some read() operation $\tau$. Let $\kappa$ be the most recent clear operation (if any exists) that completed prior to the beginning of $\tau$. Let $P$ be the set of join operations that end after $\kappa$ begins and begin no later than when the read operation ends. Then $z \leq \gamma|P|$ with probability at least $1 - 1/n^c$. □

Next, we show that if there is sufficient helping, every process is counted (even those that are slow).

*Lemma 4:* For every constant $c$, there exists a constant $\psi < 1$ (as a function of $c$) such that: Let $z$ be the value returned by some read() operation $\tau$. Let $H$ be a set of help() operations, where $|H| \geq 3c^2 \log^4 n$. Assume that there are no concurrent clear operations, and let $\kappa$ be the most recent clear operation that completed (if any exist).

If there exists a set of $p \geq T$ processes that execute at least the first write step of a pre-join operation after the clear operation $\kappa$ and prior to the first operation in $H$, and if the read operation $\tau$ begins after the last operation in $H$ completes, then $z \geq \psi p$ with probability at least $1 - 1/n^c$.

```
1  object approx-counter⟨c⟩
2    // Threshold between small & normal activity:
3    threshold T = Θ(log^4 n)
4    M : max-register⟨log n⟩, initially 0
5    // Array of activity ctrs, counting to c log^2 n:
6    C[1..log n] : basic-counter⟨{1..c log^2 n}⟩
7    // L[i][j] is value in jth leaf of the ith ctr:
8    L[1..log n][1..c log^2 n] : two-dim array of bits
9    sC : bounded-counter⟨T⟩
10
11 procedure pre-join() // Indicate intent to join.
12   // Random exponentially distributed choice:
13   Choose i ∈ {1..log n}: Pr(i) = 1/2^i
14   // Random uniformly distributed choice:
15   Choose j ∈ {1..c log^2 n}: Pr(j) = 1/c log^2 n
16   L[i,j] ← 1  // Process makes its mark.
17   return ⟨i,j⟩  // Return counter and port.
18
19 procedure join(⟨i,j⟩) // Join counter i at port j.
20   // Copy in case of concurrent clear:
21   C' ← copy of pointer to C[i]
22   M' ← copy of pointer to M
23   S' ← copy of pointer to sC
24   // If the mark is still there:
25   if L[i,j] = 1 then
26     C'.join(j)  // Join ctr i at port j.
27     // If ctr is big, then write max-register.
28     if (C'.read() ≥ c log n) then M'.write(i)
29     S'.join()
30
31 procedure check(⟨i,j⟩) return (L[i,j] = 1)
32
33 procedure read()  // Read approx counter.
34   v1 ← sC.read()
35   v2 ← M.read()
36   if v1 ≥ T then return max(v1, 2^v2 log n)
37   else return v1
38
39 // Help those that have not finished joining.
40 procedure help()
41   // Pick a random ctr i and leaf j to help join.
42   Choose i ∈ {1..log n}: Pr(i) = 1/log n
43   Choose j ∈ {1..c log^2 n}: Pr(j) = 1/c log^2 n
44   if L[i,j] = 1 then
45     C[i].join(j)
46     if (C[i].read() ≥ c log n) then M.write(i)
47
48 // The clear operation cannot be called concurrently.
49 procedure clear() // Clearing is not atomic.
50   M ← new(max-register⟨log n⟩)
51   C ← new-array(basic-counter⟨1..c log^2 n⟩)
52   sC ← new(bounded-counter⟨T⟩)
53   for i = 1 to log n, j = 1 to log^2 n do L[i,j] ← 0
```

Finally, we bound the cost of using the approximate counter:

*Lemma 5:* Suppose that after each clear operation, there are at least $\Omega(T)$ join operations before the next clear. Then each pre-join, join, check, read, and help operation has amortized cost $O(\log^2 \log n)$, with high probability. Each

clear operation has cost $O(\log^2 n)$. □

Note that we only clear the counter once $\Theta(T)$ processes have joined; otherwise, the above bounds may not hold.

## 5. Dynamic Epoch-based Counter

This section describes a dynamic counter that supports both join and leave operations. The basic idea is to divide the execution into epochs, where each epoch has an approximate counter. As long as the number of processes that have joined an epoch is much larger than the number of processes that have left an epoch, then the value returned by the counter remains a good approximation. Whenever the number of departed processes reaches a constant fraction of the processes that have joined, then a new epoch is triggered, and all the processes are awakened and instructed to join the new epoch.

There are two challenges here. First, the dynamic counter must continue to give good estimates, even as processes slowly transition from one epoch to the next. Second, as already discussed, the *very first* step of a process must be a write operation that "makes a mark." Hence, a new process must increment the counter before reading the epoch counter.

We solve both problems as follows: instead of allocating one approximate counter per epoch, we use three approximate counters, rotating the approximate counter in use for the current epoch. That is, in epoch $e$, we read from counter $e \mod 3$. When a process wants to join, it randomly chooses a counter to increment, ensuring that a constant fraction of joins update the right counter. When we transition from epoch $e$ to $e+1$, we clear counter $(e-1) \mod 3$, so we can continue to observe the final count from epoch $e$, while we transition to $e+1$.

*Joining:* A process $p$ joins the dynamic counter by calling the function join$(p, f)$, where $f$ is a callback function to be executed when the epoch ends. The first part of the join procedure chooses an approximate counter to increment (Line 17) and attempts to execute a pre-join on that counter (Lines 22–27). This pre-join is where the process performs its first write, ensuring that it can be counted immediately. The pre-join is repeated until it succeeds in making its mark in an epoch without the epoch changing. At this point the join is completed (Line 28), and the process helps each of the three counters (Line 30). Finally, the process registers the callback function to be triggered when the epoch ends. Specifically, if the process joined at epoch $e$, then the callback function is triggered when $new\text{-}epoch[e]$ toggles to true, indicating that the epoch finished, and the procedure join returns the value $e$.

*Leaving:* When a process that joined epoch $e$ exits the critical section, it executes leave$(e)$. Notice that since a leave is executed only after a critical section, there are never concurrent leave operations. The process first checks whether the epoch has advanced beyond $e$, and if so, the leave is ignored. Next, the departing process increments the *leave counter* (Line 48) which tracks the number of processes that have left. (Since there are no concurrent leave operations, we need not implement a concurrent counter.) Next, the process reads the dynamic counter (Line 49) to check how many processes are currently active. If the number of processes that have left has reached a constant fraction of the processes that have joined (Line 53), and if at least $T$ processes have left, then the epoch ends: the counter for epoch $e-1$ is cleared, the epoch is incremented, and processes are triggered to join a new epoch. Note that at least $T$ processes complete in each epoch.

*Analysis:* We now show that the dynamic counter returns a constant-factor approximation of the number of active processes. We first argue that the number of processes that take steps in epoch $e-1$ is no greater than some constant times the number of processes that complete in epoch $e$. This allows us to amortize the work done by processes in $e-1$ against processes that complete in epoch $e$. Let $P_e$ be the set of processes that execute one read/write step of Line 22 in epoch $e$.

*Lemma 6:* For every constant $c$, there exists a constant $\varepsilon < 1$ such that: If $\zeta$ is the last leave operation in epoch $e$, then at the end of $\zeta$, $Lcount[e] \geq \varepsilon |P_{e-1}|$ with probability at least $1 - 1/n^c$. □

We now show that the dynamic counter returns a value that is at least as large as a constant $\varepsilon$ times the number of processes that completed a join operation in that epoch.

*Lemma 7:* For every constant $c$, there exists a constant $0 < \varepsilon < 1$ such that: For every read operation $\tau$ in epoch $e$, where $z \geq 0$ is the value returned by $\tau$, if there exists a set $P \subseteq P_e$ of at least $48cT$ processes that completed a join in epoch $e$ prior to the beginning of $\tau$, then $z \geq \varepsilon |P|$. □

Next, we show that the counter cannot grow too big: its value is bounded by the number of processes that have joined in the most recent three epochs. This follows since the counters are cleared every three epochs.

*Lemma 8:* For every constant $c$, there exists some constant $\gamma_1 > 1$ such that: For every read operation $\tau$ in epoch $e$ that returns value $z$, if $P$ is the set of processes that take one step of a join in epochs $e-2$ or $e-1$ or $e$, where the join begins prior to the end of $\tau$, then $z \leq \gamma_1 |P|$. □

Finally, we show that the value returned by the dynamic counter is at most a constant factor greater than the number of active processes. This lemma relies on two facts: First: at the end of each leave operation, the value of the leave counter is at most a constant fraction of the number of active processes. This is important, as it ensures that not too many processes leave during an epoch, and hence we can prove a second claim: every read operation returns a value at most some constant times the number of processes active at the end of the operation. The first claim itself depends on the second, however, as the decision to end an epoch depends on the value of the counter (Line 53). We must also account for when the number of processes falls below the threshold $T$. Let $A_\zeta$ be the number of active processes at the end of leave

```
1  object dynamic-counter⟨c⟩
2    // Threshold between small and normal activity.
3    threshold T = Θ(log^4 n)
4    C[0..2] : approx-counter⟨c⟩
5    // The epoch is updated by a leaving process.
6    epoch : global integer, initially 1
7    // Count # processes that leave in epoch e:
8    Lcount[1..n] : array of integers, initially all 0
9    // During epoch e, new-epoch[e − 1] = true.
10   new-epoch[1..n] : array of bits, initially false
11   // Proc. p stores epoch it joined in last-epoch[p].
12   last-epoch[1..n] : array of int, initially all 0
13
14 // on-new-epoch is called when epoch ends.
15 procedure join(p, on-new-epoch)
16   // Choose approx. counter to increment.
17   Choose i ∈ {1..3} such that: Pr(i) = 1/3
18   done ← false
19   while done = false do
20        // Port b of counter a marked.
21        // First RMR occurs here in pre-join.
22        ⟨a, b⟩ ← C[i].pre-join()
23        last-epoch[p] ← epoch   // Read epoch.
24        // Check that ctr has not been cleared and
25        // that the epoch has not changed.
26        done ← C[i].check(⟨a, b⟩)  and
27             last-epoch[p] = get-epoch()
28   C[i].join(⟨a, b⟩)   // Finish join port b, ctr a.
29   // Help increment counters
30   C[0].help(); C[1].help(); C[2].help()
31   // Monitor last-epoch[p].
32   // Call on-new-epoch when epoch changes.
33   register(when(new-epoch[last-epoch[p]]=true))
34        do on-new-epoch(last-epoch[p] + 1))
35   return(last-epoch[p])   // Return epoch number.
36
37 procedure read()
38   e ← epoch   // Read current counter.
39   v ← C[e mod 3].read()
40   // If epoch changed, return -1.
41   if epoch = e then return v
42   else return −1
43
44 // The leave operation cannot be called concurrently.
45 procedure leave(e) // Leave after exiting critical sec.
46   if epoch = e then
47        // Increment count of departed procs:
48        Lcount[epoch] ← Lcount[epoch] + 1
49        total ← read()   // Read dynamic counter.
50        // If # departures is above threshold,
51        // and # departures is a large-enough
52        // fraction of arrivals, epoch ends.
53        if (Lcount[epoch] > λ · total)  and
               (Lcount[epoch] ≥ T)  then
54             // Clear counter for previous epoch.
55             C[(epoch − 1) mod 3].clear()
56             epoch ← epoch + 1   // Increment epoch.
57             // Trigger wake-up of all procs.
58             new-epoch[epoch − 1] ← true
59
60 procedure get-epoch()
61   return epoch
```

$\zeta$ in epoch $e$, and let $A_\tau$ be the number of active processes at the end of $\tau$. Note that $\lambda < 1$ is defined in Line 53, and its precise value is fixed to $\Theta(1/\gamma_1)$ in the proof.

*Lemma 9:* For every constant $c$, there exists a constant $\beta \geq 1$ such that for every epoch $e$:

1) For every leave operation $\zeta$ in epoch $e$: $Lcount[e] \leq \max(4\lambda\beta|A_\zeta|, T)$ at the end of $\zeta$, with probability at least $1 - 1/n^c$.
2) For every read operation $\tau$ in epoch $e$ that returns value $z$, then $z \leq \beta \max(|A_\tau|, T)$, with probability at least $1 - 1/n^c$. □

## 6. Mutual Exclusion Algorithm

We now give the mutual exclusion algorithm. To ensure safety, the protocol guards the critical section with a *lock*. A process can enter the critical section only after writing its identifier to the lock with a CAS operation. We use a dynamic counter $C$ to track the number of active processes and to define the epoch structure. When each competing process $p$ finishes joining epoch $e$, it attempts to find a slot in a dense array $A[e]$. The array $slot[p]$ stores the slot in the array $A[e]$ that $p$ is currently holding. For each epoch $e$, we also maintain a second approximate counter $joinCount[e]$, which counts the number of processes that have successfully found a slot in the array $A[e]$. (The difference $C$ minus $joinCount[e]$ indicates how many processes have begun but not yet finished joining.)

For when there are only a small number of processes, each epoch $e$ also has a deterministic *small* mutual exclusion object $sMutex[e]$ with only $\Theta(T)$ ports (i.e., it costs each process $O(\log T)$ to use). A bounded counter $sC[e]$ (with max value $\Theta(T)$) assigns the first $\Theta(T)$ processes to join an epoch to ports.

*Competing for the Critical Section:* When a process $p$ is first activated or awakened to join a new epoch, $p$ executes the compete procedure. The first step is to clear all events registered in prior epochs. Next, $p$ executes a join on the dynamic counter (Line 24) with the call-back function compete as a parameter. This callback function indicates that when the next epoch begins, the process $p$ should call compete again. Note that this is the first place where process $p$ performs an RMR after awakening, and critically, that the RMR is a write to memory that makes a mark, allowing the other processes to observe $p$'s existence.

Next, process $p$ loops (Lines 28–31): it reads the dynamic counter $C$, and attempts to claim (via a CAS) a random location in the array $A[e]$, selecting within a subarray based on the value returned by $C$, but always of size $\Omega(T)$. The density of $C$ is determined by the accuracy of the counter $C$: ideally, a constant-fraction of the slots in the array $A[e]$ are full. This allows $p$ to readily find a free slot. Next, in Line 34, the process $p$ increments the counter $joinCount[e]$, indicating that it has successfully joined array $A[e]$.

```
1  object mutual-exclusion⟨c⟩
2    constant T = Θ(log^4 n)
3    constant β, as defined in Lemma 9
4    constant δ, as defined in Lemma 2
5    constant ε, as defined in Lemma 7
6     // Lock to guard critical section.
7    lock : process identifier, initially zero
8    C : dynamic-counter⟨c⟩, dynamic counter
9     // In epoch e, spin in A[e] until awakened:
10   A[1..n][1..Θ(n)] : 2D array
11    // Slot in A where p is currently spinning:
12   slot[1..n] : array of integers, one per process
13    // Counts # processes that have finished joining:
14   joinCount[1..n] : approx-counter⟨c⟩, one per epoch
15    // Used to claim a port in small mutex instance:
16   sC[1..n] : bounded-counter⟨(48cβ/ε)T⟩
17    // Small mutex instances:
18   sMutex[1..n] : array of deterministic mutual exclusion
       instances of size (96cβ/ε)T
19
20 procedure compete(p)
21    // Stop interrupts from earlier epoch events.
22   clear-registered-events(p)
23    // Join dynamic ctr: callback function is compete.
24   e ← C.join(p, compete(p))
25   done ← false
26   while done = false do
27        // Read dynamic counter.
28       v ← max(C.read(), (48c/ε)T)
29       Randomly choose slot[p] ∈ {1..(4/ε)v}
30        // Claim slot in array A.
31       done ← CAS(A[e, slot[p]], 0, p)
32    // Count procs that finished joining the array.
33   ⟨a, b⟩ = joinCount[e].pre-join()
34   joinCount[e].join(⟨a, b⟩)
35    // First Θ(T) processes join the small mutex.
36   s ← sC[e].join()
37   if s ≥ 0 then
38       sMutex[e].join(s, small-mutex-win(p, e))
39    // Register an interrupt if a slot in A is updated.
40   register(when(A[e, slot[p]] changes))
41        do mutex-win(p, e))
42   CAS(lock, 0, p)   // Try to claim lock.
43    // If p gets lock, attempt to enter critical sec.
44   if lock = p  then  mutex-win(p,e)
```

```
45 procedure small-mutex-win(p, e)
46   CAS(lock, 0, p)   // Try to claim lock.
47    // If p gets lock, attempt to enter critical sec.
48   if lock = p  then  mutex-win(p,e)
49   else   // If lock frees, try again.
50   register(when (lock = 0)) do small-mutex-win(p, e))
51
52  // Try to enter critical section for epoch e.
53 procedure mutex-win(p, e)
54   CAS(lock, 0, p)
55   if (lock = p)  and (C.getEpoch() = e)  then
56        // Stop anything that can interrupt process.
57       clear-registered-events(p)
58       Execute critical section.
59       sMutex[e].leave()   // Exit small mutex instance.
60       C.leave(e)   // Leave dynamic counter.
61       A[e, slot[p]] ← 0   // Clear array slot.
62       CAS(lock, p, 0)   // Release lock.
63       done ← false   // Find process to handoff to.
64       while done = false do
65           v1 ← C.read()
66           v2 ← joinCount[e].read()
67            // If small # of participants:
68           if v1 ≤ (32βγ/δ)T then  done ← true
69            // If we know ≥ 1 processes are joining: else
             if v1 ≥ 2(β/δ)v2 then  done ← true
70           else
71               Randomly choose i ∈ {1..Δv2}.
72               x ← A[e, i]
73               if x>0 then done ← CAS(A[e, i], x, p)
```

The next part of the compete procedure copes with the case when there are a small number of processes active in epoch $e$. Process $p$ increments the small counter $sC$, and if it is one of the first $O(T)$ processes to do so (i.e., if it gets back a value $s \geq 0$) then it competes in the small mutual exclusion instance for epoch $e$, i.e., $sMutex[e]$. The function small-mutex-win$(p, e)$ is passed as the callback function to be executed if $p$ wins the $sMutex[e]$ instance. If $p$ wins, it continually tries to get the lock until it succeeds or a new epoch begins. At any given time, at most one process has won the $sMutex$ instance and is waiting for the lock.

In the last part of the compete procedure, process $p$ registers an event, i.e., to call mutex-win if some other process awakens it by modifying its slot in the array $A[e]$. Process $p$ then checks whether the $lock$ is available, and if so, it tries to acquire it and execute mutex-win. After trying to acquire the lock, process $p$ can safely spin, waiting for either a new epoch or to be awakened via the array $A$.

*On Winning the Lock:* When process $p$ is awakened via the array $A$, it executes mutex-win. The first action by $p$ is to try to acquire the $lock$. If it fails, then it continues spinning. If it succeeds, and if the epoch has not changed, then it clears all registered events and enters the critical section. (Note that up until this point, it may be interrupted by other events, e.g., a new epoch or a new mutex-win.)

When $p$ exits the critical section, it leaves the small mutual exclusion instances $sMutex[e]$, it leaves the counter $C$, and it departs from the array $A[e]$. (Note that it does not matter in which way $p$ won the critical section.) It then releases the lock on Line 62, allowing others to enter the critical section.

The remainder of the mutex-win procedure ensures that some other process will later enter the critical section. If the number of processes in epoch $e$ is small, i.e., they are all contained in $sMutex[e]$, then there is no need to do any further work. (This is checked on Line 68.) On the other hand, if there are processes that have entered, but not yet finished joining the array $A[e]$ (and not yet begun to spin), then process $p$ can safely exit. (This is checked on Line 69.)

Otherwise, process $p$ must find some process in the array

and wake it. A key technical challenge is ensuring that this array remains dense, and hence that spinning processes are easy to find, even as some processes may not yet have joined the array $A[e]$, and other processes may have already left the system. Fortunately, if there are $\Omega(T)$ processes that have joined epoch $e$, and if all of those processes have completed joining the array $A[e]$, then we can be sure that the array is dense, and hence with constant probability, process $p$ will find a spinning process (Lines 71–73).

*Analysis:* Mutual exclusion follows trivially from the use of a lock to protect the critical section.

*Theorem 10:* For every execution, no two processes enter the critical section at the same time. □

The second key claim is that there is no deadlock.

*Theorem 11:* Every process eventually enters the critical section, with high probability. □

We now examine the performance of the protocol.

*Theorem 12:* In each execution there are $O(n \log^2 \log n)$ RMRs, with high probability. □

## 7. Conclusion

We have presented a new mutual exclusion algorithm with amortized $O(\log^2 \log n)$ RMRs per process, with high probability.

We have achieved exponentially smaller bounds by weakening the model of in two ways. First, we assumed an oblivious adversary, whereas Hendler et al. [14] assume an adaptive adversary. It has been conjectured that the $O(\log n / \log \log n)$ RMR bound is the best possible for an adaptive adversary [18], at least for algorithms that are starvation-free. If so, then our choice of a weaker adversary seems fundamental. It would be interesting to explore intermediate adversaries (see, e.g., [7], [9]).

Second, we ensure only that all processes enter the critical section with high probability (rather than with probability 1). This weakening does not seem fundamental. By detecting when the counter's value is too big or too small, it may be possible to avoid deadlock in all cases.

It would also be interesting to consider other *local spinning* models. It seems likely that the results here extend to the DSM (dynamic shared memory) model, but there are subtle differences to resolve.

## References


[1] J. H. Anderson and Y.-J. Kim, "Fast and scalable mutual exclusion," in *13th International Symposium on Distributed Computing*, 1999, pp. 180–194.

[2] ——, "An improved lower bound for the time complexity of mutual exclusion," *Distributed Computing*, vol. 15, no. 4, pp. 221–253, 2002.

[3] ——, "Nonatomic mutual exclusion with local spinning," in *Proceedings of the Twenty-First ACM Symposium on Principles of Distributed Computing (PODC)*, 2002, pp. 3–12.

[4] J. H. Anderson, Y.-J. Kim, and T. Herman, "Shared-memory mutual exclusion: major research trends since 1986," *Distributed Computing*, vol. 16, no. 2-3, pp. 75–110, 2003.

[5] J. Aspnes, H. Attiya, and K. Censor, "Max registers, counters, and monotone circuits," in *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2009, pp. 36–45.

[6] J. Aspnes and K. Censor, "Approximate shared-memory counting despite a strong adversary," in *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2009, pp. 441–450.

[7] H. Attiya and K. Censor, "Lower bounds for randomized consensus under a weak adversary," in *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2008, pp. 315–324.

[8] H. Attiya, D. Hendler, and P. Woelfel, "Tight RMR lower bounds for mutual exclusion and other problems," in *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2008, p. 447.

[9] Y. Aumann and M. A. Bender, "Efficient low-contention asynchronous consensus with the value-oblivious adversary scheduler," *Distributed Computing*, vol. 17, no. 3, pp. 191–207, 2005.

[10] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Comm. ACM*, vol. 8, no. 9, p. 569, 1965.

[11] R. Fan and N. A. Lynch, "An Omega(*n* log *n*) lower bound on the cost of mutual exclusion," in *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2006, pp. 275–284.

[12] W. M. Golab, V. Hadzilacos, D. Hendler, and P. Woelfel, "Constant-RMR implementations of CAS and other synchronization primitives using read and write operations," in *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2007, pp. 3–12.

[13] W. M. Golab, L. Higham, and P. Woelfel, "Linearizable implementations do not suffice for randomized distributed computation," in *Proceedings of the 43rd Symposium on Theory of Computing (STOC)*, 2011.

[14] D. Hendler and P. Woelfel, "Randomized mutual exclusion in O(log N / log log N) RMRs," in *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2009, pp. 26–35.

[15] ——, "Adaptive randomized mutual exclusion in sub-logarithmic expected time," in *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2010, pp. 141–150.

[16] Y.-J. Kim and J. H. Anderson, "A time complexity bound for adaptive mutual exclusion," in *Proceedings of the 15th International Conference on Distributed Computing (DISC)*, 2001, pp. 1–15.

[17] L. Lamport, "A new solution of dijkstra's concurrent programming problem," *Commun. ACM*, vol. 17, pp. 453–455, August 1974.

[18] P. Woelfel, Private communication, 2010.

[19] J.-H. Yang and J. H. Anderson, "A fast, scalable mutual exclusion algorithm," *Distributed Computing*, vol. 9, no. 1, pp. 51–60, 1995.