# A Fair Fast Distributed Concurrent-Reader Exclusive-Writer Synchronization

Theodore Johnson and Hankil Yoon
Dept. of CISE, University of Florida
Gainesville, FL 32611-2024
ted@cis.ufl.edu, hyoon@cis.ufl.edu

May 8, 1996

**Abstract**

Distributed synchronization is needed to arbitrate access to a shared resource in a message passing system. Reader/writer synchronization can improve efficiency and throughput if a large fraction of accesses to the shared resource are queries. In this paper, we present a highly efficient distributed algorithm that provides FCFS concurrent-reader exclusive-writer synchronization with an amortized $O(\log n)$ messages per critical section entry and $O(\log n)$ bits of storage per processor. We evaluate the new algorithm with a simulation study, comparing it to fast and low-overhead distributed mutual exclusion algorithms. We find that when the request load contains a large fraction of read locks, our algorithm provides higher throughput and a lower acquire time latency than is possible with the distributed mutual exclusion algorithms, with a small increase in the number of messages passed per critical section entry. The low space and message passing overhead, and high efficiency make the algorithm scalable and practical for implementation. The algorithm we present can easily be extended to give preference to readers or writers.

## 1 Introduction

Distributed mutual exclusion is often implemented by exchanging a *token*, which represents the privilege of accessing a resource [16, 17, 15, 11, 21, 3, 6, 5]. Reader-writer (RW) synchronization admits multiple concurrent readers into the critical section (CS) while allowing only one writer to exclusively access the CS. If readers constitute a large fraction of the request stream, then RW synchronization can increase parallelism and eliminate synchronization bottlenecks [22].

In this paper, we present a distributed concurrent-reader exclusive-writer synchronization algorithm based on the path compression technique of Chang, Singhal, and Liu (CSL) [3] which requires $O(\log n)$ bits of storage per processor (to uniquely identify $n$ processors) and an amortized of $O(\log n)$ messages per CS entry. The low space and message passing overhead, and high efficiency make the algorithm scalable and practical for implementation. The processors synchronize

by sending and interpreting messages. We assume that every message that is sent is eventually received. We implemented a simulation of the algorithm and made a performance study against the fixed tree (FT) algorithm [17] and the CSL algorithm [3].

Section 2 discusses related works and presents a brief introduction to the FT algorithm and the path compression technique. Section 3 presents the data structure defined by the algorithm, and describes how the algorithm works. A short discussion on the correctness of the algorithm and theoretical issues are given in Section 4. We provide a performance analysis in Section 5 to show how our algorithm outperforms the FT and the CSL algorithms using various performance measures that result from the simulation study. The paper concludes with Section 6.

## 2 Background

Considerable attention has been paid to the problem of distributed synchronization. Recent works have developed fast and low-overhead distributed mutual exclusion algorithms which require only $O(\log n)$ messages per critical section entry and $O(\log n)$ bits of storage per processor.

Raymond [17] proposed a simple hierarchical structure imposed on processors to reduce the number of messages that are passed (see Figure 1). Each processor has a fixed set of neighbors. A processor points to the neighbor that is closer to the current token holder with the variable current_dir. When a processor receives a request for the token, it fowards the request to the neighbor indicated by current_dir and records the requesting neighbor in a FIFO queue. Subsequent requests do not need to be forwarded because the token has already been requested. The record of the requesting processors forms a return path to pass the token to the next processor to enter the critical section.

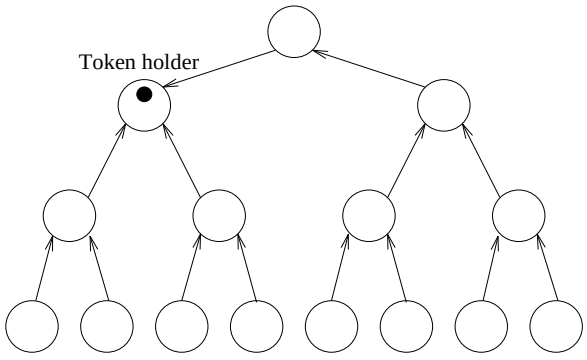


Figure 1: An example of the fixed tree structure

Chang, Singhal, and Liu [3] present a distributed mutual exclusion based on *path compression*. Every processor stores a pointer current_dir which points towards the last processor waiting for the mutual exclusion token. The waiting processors form a waiting list using pointer next (the head of the list is the token holder). So, the current_dir pointers point towards the last processor in the list formed by the next pointers (or towards the token holder if there is no waiting list). The logical structure changes dynamically. When a processor receives a request for a token, it forwards the request in the direction of current_dir (with some exceptions for special cases). Since the requester soon will be added to the end of the waiting list, the processor sets current_dir to point to the requester. Since a processor that handles a request message becomes close to the end of the waiting list, request paths are compressed by the actions of handling a request. Hence, this technique is called *path compression*. While the upper bound of the number of messages required to request the critical section can be $O(n)$ in the worst case, the amortized number of messages per critical section entry is $O(\log n)$.

While these distributed synchronization algorithms are fast and efficient, they do not provide reader/writer synchronization, making them inefficient when a large fraction of requests are queries. Several efficient (e.g., contention-free) shared-memory reader/writer synchronization algorithms have been proposed. Mellor-Crummey and Scott [14] developed a scalable reader/writer lock (an MCS lock), which they implemented on the BBN TC2000. Their locks depend on the rich set of atomic read-modify-write operations provided by the BBN TC2000, and require three global variables. Krieger *et al.* [8] proposed an improvement over the MCS locks, but it still depends on an atomic read-modify-write operation and a global variable. Translating these algorithms into message passing algorithms requires the use of a centralized management processor to supply the function of the atomic read-modify-write operations on the global variables.

# 3 Description of the algorithm

The distributed concurrent-reader exclusive-writer (dCREW) algorithm that we present is an extension of the CSL path-compression distributed mutual exclusion algorithm. As in the CSL algorithm, two lists are threaded through the processors: the *path list*, defined by the current_dir pointers, and the *waiting list*, defined by the next pointers (see Figure 2). Consecutive readers in the waiting list can all enter the critical section concurrently.

**Data structure** Participants in the dCREW algorithm (particularly those in the waiting list) need to store information about the state of the synchronization. We provide here a list of the variables used and their meaning.

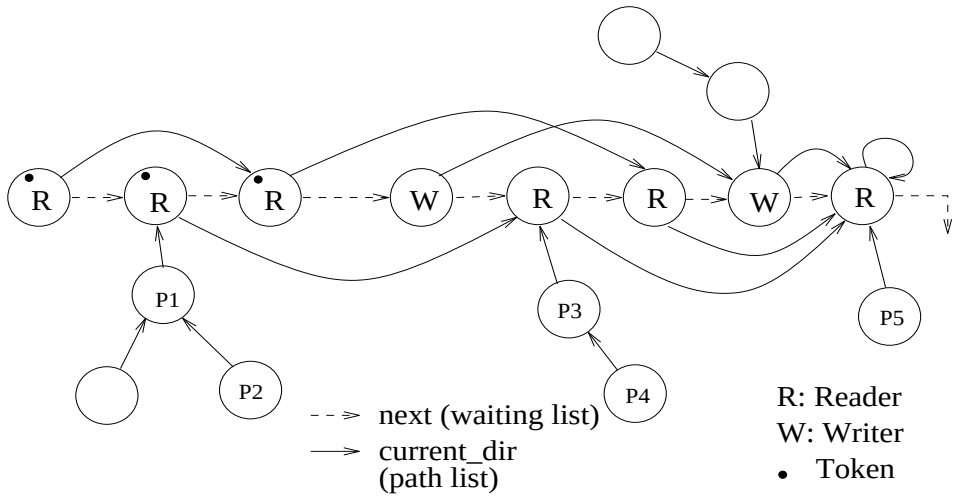


Figure 2: An example of the processor structure in the dCREW algorithm

1. IsRequesting: True if the processor has requested the token and is waiting for it.
2. Token_hldr: True if the processor is holding the token.
3. Incs: True if the processor is in the CS. Note that Token_hldr must be be True if Incs is True.
4. next: The ID of the next processor in the waiting list. This variable is NIL if there is no next processor, or if the processor is not requesting or using the token.
5. current_dir: A processor ID that represents the current best guess for the last processor in the waiting list (or the token holder if there is no waiting list).
6. class: The value (one of none, reader, or writer) indicates the type of request that the processor is making.
7. successor: The value (one of none, reader, or writer) represents the type of request that the next processor has made. The last active reader must have none for this variable.
8. release_needed: The number of release tokens that a reader needs to receive before completion (0 or 1).

The current_dir pointers form the path list (for the collection of all processors, actually a tree). For a given processor, the path list leads to where the processor should enter the waiting list. If a waiting list exists, the path list leads to the end of the waiting list, else the path list leads to the current token holder.

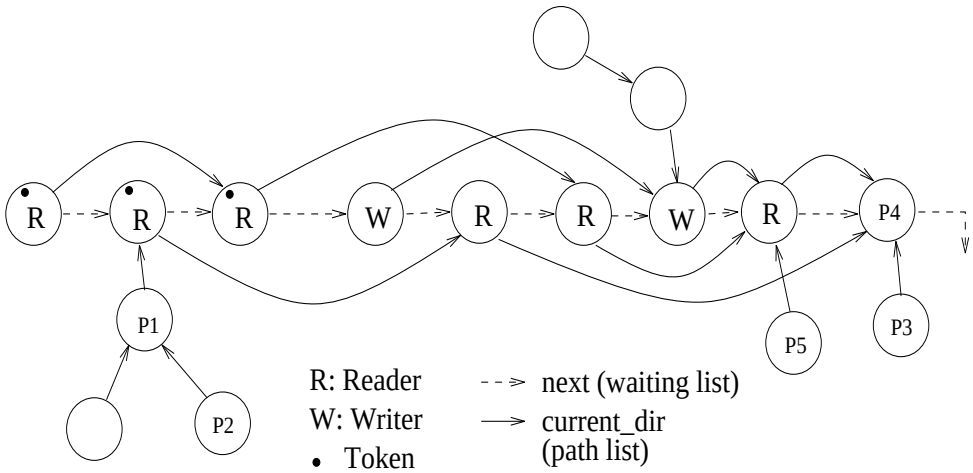


Figure 3: The new processor structure after $P_4$ makes a request

The waiting list is maintained through the next pointers, which are non-NIL only at the processors that either holding or requesting the token. A maximal consecutive sequence of readers in the waiting list are permitted to enter the critical section concurrently. When the first reader in the sequence receives the token, it replicates the token and passes the token to the subsequent reader. This process continues until all readers in the sequence have entered. A processor that requests exclusive access to the resource does not replicate the token. To control the token replication, each processor in the waiting list must know the type of its request (class) and also the type of the request of the processor pointed to by next (in successor).

All processors in the maximal sequence of readers must release the token before the token can be given to a writer. To avoid complicated distributed list manipulation, we require that before a processor can release a read lock, all processors that preceded it in the waiting list must have also released their lock. Permission to release the lock is expressed as a *release token*. The first reader in the sequence may release its lock at any time. The second reader in the sequence may release its lock when it has completed its critical section, and it has received the release token from the first reader. Similarly, the $k$th reader in the sequence can release its lock only after receiving a release token from the $k-1$st reader. The number of release tokens expected by a reader are recorded in release_needed (0 or 1).

Preventing the $k$th reader from releasing its lock until the $k-1$st reader has released its lock can cause a performance degradation (i.e., *reader stall*). The application thread that releases the lock can continue processing. However, a stalled reader processor is blocked from requesting another lock until the previous lock record is removed from the waiting list. We also discuss a technique

that removes reader stall at the cost of a more complex algorithm and increased space overhead.

**Message types** There are two kinds of messages in the algorithm: *request* and *token*. A request is passed to the processor pointed to by current_dir until it reaches the free token or the end of the waiting list. There are two parameters for a request message: requester (the ID of the requester), and requester class which indicates the type of request (reader or writer).

There are two types of tokens: the *grant* token and the *release* token. The grant token gives the receiver permission to enter the critical section, and the release token gives the receiver permission to remove its lock control block from the waiting list. One parameter is passed along with the token: token_type which indicates the number of release tokens the receiver should expect.

**Semantics of notations** A processor can send a message of type action to processor destination with parameters parameters by executing following statement:

send(destination, action; parameters)

The message passing is assumed to be reliable, but the latency of the underlying communication network is unpredictable. In addition to sending messages, processors need to be able to receive messages as part of the protocol that they execute. We generalize the receipt of a message to the receipt of an *event*. An event can not be handled until a thread declares that it will process the event, and the event is buffered until it is handled. If a processor is able to handle the event at any time, it must execute a thread dedicated to handling the event. A processor declares that it is waiting for events $A_1$, $A_2$, ...,$A_n$ by executing following code:

```
wait for A1,A2,...,An
    A1(source; parameters):
        code to handle A1
    ...
    An(source; parameters):
        code to handle An
```

When a processor $p$ executes send($q$, $A_1$; parameters) and processor $q$ executes the above code, then $q$ will eventually process the message sent by $p$. The variable source contains the processor name $p$, and the parameters that $p$ sent will be unpacked by $q$. The semantics of this construction are similar to the select system call used with Berkeley sockets.

### 3.1 Requesting the token

When a processor wants to enter the CS operation, it sets next to NIL, and sends a request message to the processor pointed to by current_dir. An exception occurs if the processor already has the token, in which case the processor is immediately allowed to enter the CS. When a processor receives a request, it sets current_dir to point to the requester. This action has an important

When a processor that is not holding the token receives a request message, it passes the request to current_dir. If the processor is the last processor in the waiting list (with next equal to NIL), it records requester's class in the successor variable and sets next to point to the requester, thus making the requester the last processor in the waiting list. If the processor at the end of the waiting list is a reader that holds the token and the requester is also a reader, then the requester can also enter the critical section. So the processor sends a grant token to the requester.

Figure 3, which evolves from Figure 2 after $P_4$ makes a request, shows an example of how the logical structure of the processors changes dynamically.

### 3.2 Releasing the token

After a processor finishes its access to the resource, it releases the token to the next processor in the waiting list (recall that readers must receive a release token). If there is no processor in the waiting list, the processor keeps the token.

A waiting writer enters the critical section after receiving the token and releases the token after exiting the critical section. A reader uses a more complex protocol, because of concurrent reader access to the shared resource. Recall that readers use two tokens, the grant token and the release token. A reader that has exited the critical section must receive a release token before it can remove its lock control block from the waiting list.

We represent the grant and release tokens implicitly – the first token that a reader receives is the grant token, and the second is the release token. When a waiting reader receives a grant token, the grant token contains the number of release tokens that must be received before the lock can be released (in the parameter token_type that is attached to the token). A reader at the head of the sequence of readers does not need release token, but all of the other readers do need a release token. A reader will be at the head of the sequence if it follows a writer, or if the reader's request arrives at a processor with a free token. In these cases, a 0 will be sent as the token_type, else a 1 will be sent as the token_type. To send a release token, a reader sends a token message with token_type set to 1. Thus, a release token is the same as a grant token that requires a release token, and the algorithm does not require FIFO message channels.

```
Request_CS_Read()
{

    wait until (release_needed eq 0);  /* reader stall */

    IsRequesting = true;

    class = READER;                    /* initialize own data structure */
    successor = none;
    next = nil;

    if Token_hldr
        current_dir = self;            /* best guess about the holder */
        release_needed = 0;            /* the first and only active reader */
    else
        send(current_dir, REQUEST; self, class);
        current_dir = self;            /* best guess about the holder */
        wait until (Token_hldr eq true);
        /* will be the last reader */

    IsRequesting = false;          /* got the token */
    Incs = true;                   /* start the CS operation */
}

Request_CS_Write()
{

    wait until (release_needed eq 0);  /* reader stall */

    IsRequesting = true;

    class = WRITER;                    /* initialize own data structure */
    successor = none;
    next = nil;

    if Token_hldr
        current_dir = self;                /* best guess about the holder */
    else
        send(current_dir, REQUEST; self, class);
        current_dir = self;            /* best guess about the holder */
        wait until (Token_hldr eq true);

    IsRequesting = false;          /* got the token */
    Incs = true;                   /* start the CS operation */
}
```

Figure 4: Primitive routines of the dCREW algorithm to request the token

```
Release_CS_Read()
{
    wait until (release_needed eq 0);       /* wait for predecessors to finish */
                                    /* However, the application can proceed */
    if next ≠ nil                           /* has waiting list or successor */
        send(next, TOKEN; 0);               /* either release or grant token */
        next = nil;
        Token_hldr = false;
    /* else just keep the token with current_dir = self */
    /* and Token_hldr = true */

    successor = none;
    Incs = false;
}

Release_CS_Write()
{
    if next ≠ nil                       /* pass the token to the waiting processor */
        send(next, TOKEN; 0);           /* grant token */
        next = nil;
        Token_hldr = false;
    /* else just keep the token */

    successor = none;
    Incs = false;
}
```

Figure 5: Primitive routines of the dCREW algorithm to release the token

```
Monitor_CS()
{
    loop forever
        wait for a REQUEST or a TOKEN message;

        REQUEST(requester, req_class):      /***** receives a request *****/
            if Token_hldr
                if Incs                 /* active reader/writer or stalled reader */
                    if next eq nil
                        /* active reader or writer at the end of the waiting list */
                        if req_class eq READER and class eq READER
                            /* pass the grant token */
                            send(requester, TOKEN; 1);
                        next = requester;
                        successor = req_class;
                    else          /* an active mid-reader or waiting list exists */
                        send(current_dir, REQUEST; requester, req_class);
                else                /* was in idle state */
                    send(requester, TOKEN; 0); /* pass the grant token */
                    Token_hldr = false;
            else if IsRequesting and next eq nil
                next = requester;         /* the last waiter */
                successor = req_class;
            else          /* mid-waiter or non-requester - pass over */
                send(current_dir, REQUEST; requester, req_class);

            current_dir = requester;  /* the best current guess about the holder */

        TOKEN(token_type):    /***** receives the token *****/
            if Incs and class eq READER     /* active reader, so its a release token */
                release_needed=0;   /* end reader stall */
            else if IsRequesting        /* waiting reader or writer */
                if class eq READER                /* waiting reader */
                    release_needed = token_type;/* exit count */
                    if successor eq READER      /* propagate the grant token */
                        send(next, TOKEN; 1);

                /* else waiting writer (just enter into the CS) */

            Token_hldr = true;
}
```

Figure 6: Event handler routine of the dCREW algorithm

**Cascading stall** A reader is *stalled* if it wishes to release its lock, but it has not yet received a release token. Because the release token must be passed from reader to reader in sequence, one slow reader can stall many fast readers, increasing critical section execution times and decreasing parallelism.

We observe that a stalled reader does not need to block the application thread at the release time. However, the application cannot request a write lock until the previous read lock has been released (a new read lock can be granted immediately). Thus, a stalled reader can block the application at request time. If the time between requests is large compared to the critical section execution time, delays due to stalled requests are negligible. If not, processors can use two lock control blocks – if one is stalled then the other is used to request the lock.

# 4 Theoretical issues

In this section, we give some intuitive arguments for correctness. We loosely refer to *events* as occurring at a point in *time*. While global time does not exist in an asynchronous distributed system, we can view the events in the system as being totally ordered using Lamport's timestamps [10], and view a point in time as being a consistent cut [2].

We note that all processors that are not requesting the token lie on a path that leads to a processor that either holds or is requesting the token. This property can be seen by induction. It is required that the property initially hold for correctness. The property then can change if a processor modifies its current_dir, or if the processor it points to changes its state. A processor that is not requesting the token will change its current_dir pointer if it receives a request. But then, it points to the most recent requesting processor. A processor can change its state from non-requesting to requesting, but the property still holds since it points to itself. Finally, a processor can change its state from holding to non-requesting, but after changing the state, the processor points to the new token holder or to a requesting processor.

The token is not lost because it is only released to a processor in the waiting list. In case of concurrent readers, the token is actually released to a processor in the waiting list by the *last* concurrent reader. We can view that the other concurrent readers simply notify the last reader of the end of their CS operation by the release token.

The number of messages per critical section entry is composed of the number of messages for the request, and the number of messages for the token. Previous analyses show that the amortized number of hops to find the end of the waiting list is $O(\log n)$ where $n$ is the number of processors in the system [4]. Either 1 or 2 tokens are passed per critical section entry. So, the amortized number of messages per critical section entry is $O(\log n)$, and the number of messages required by

the dCREW algorithm is within 1 of the number of messages required by the CSL algorithm.

# 5 Performance analysis

In order to precisely quantify the performance of the dCREW algorithm, we made a simulation study of the dCREW algorithm, the FT algorithm, and the CSL algorithm. The simulator modeled a set of processors that communicate through message passing. All delays are exponentially distributed. The simulation parameters are the number of processors, the message transit delay (with mean value of 1 unit of time), the message processing delay (1 unit), the time between releasing the token and requesting it again (the inter-access time, varied), the time that the token is held once acquired (the release delay, 10 units), and the ratio of readers to a writer (1, 2, 4, and 9). The FT algorithm uses a nearly-complete binary tree as depicted in Figure 1.

We ran the simulator for varying number of processors and varying loads, which we define to be the product of the number of processors $n$ and the release delay $C$ divided by the inter-access time $R$ (i.e., $load = nC/R$). Note that it is meaningful to have a load larger than 100%. For each run, we executed the simulation for 100,000 CS entries. We collected a variety of statistics including: the amount of time to finish the simulation (which captures the time overhead of running the protocol), the number of messages sent, the average waiting time to acquire the token after request, the sum of the times that the token is in use, and the sum of the times that the token is being requested or in use.

The performance of the algorithm depends on the ratio of readers to writers, $R_{rw}$. The dCREW algorithm improves performance (as measured by response time and throughput) by permitting parallel access to the critical section. As $R_{rw}$ becomes large, the advantage increases. However, we can expect that the dCREW algorithm requires slightly more messages than the CSL algorithm. As was shown in a previous study [6], the FT algorithm requires significantly more messages than the CSL algorithm under light loads, but fewer messages under heavy loads. Given the similarities between the CSL and the dCREW algorithm, we expect that a similar comparison can be made between the FT and the dCREW algorithms.

For convenience, we will use the term *lock* interchangeably with the term *token* in the following sections. The number after 'dCREW' in the figures is the ratio of readers to a writer in the experiment.

## 5.1 Messages per Critical Section

As the first measure of the performance, we plot the number of messages sent per CS entry against the number of participating processors. Every processor issues requests at the same rate, and the

load ($nC/R$) varies between 50% and 200%. The results of this measure are plotted in Figures 7 and 8.

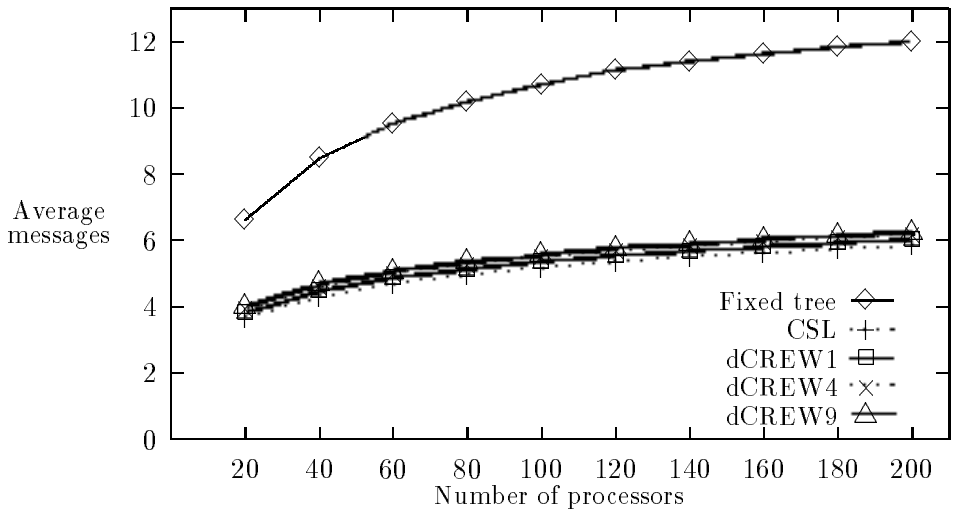


Figure 7: Average messages per CS entry with 50% load

The dCREW requires slightly more messages per critical section entry than does the CSL algorithm. However, this difference is so small as to be hard to distinguish in the charts. Under low load, the dCREW algorithm needs fewer messages than the FT algorithm, but more messages than the FT algorithm under a high load. This behavior occurs because the fixed structure of the FT algorithm lets it terminate request messages early (as was shown in [6]). The number of messages per critical section for the dCREW algorithm varies little as the ratio of readers to a writer varies.

### 5.2 Token acquisition time

The average time between requesting and acquiring the token is shown in Figures 9 through 11 for loads varying between 50% and 200%. A lower token acquisition time translates into lower lock overhead and thus faster response times in the parallel computation.

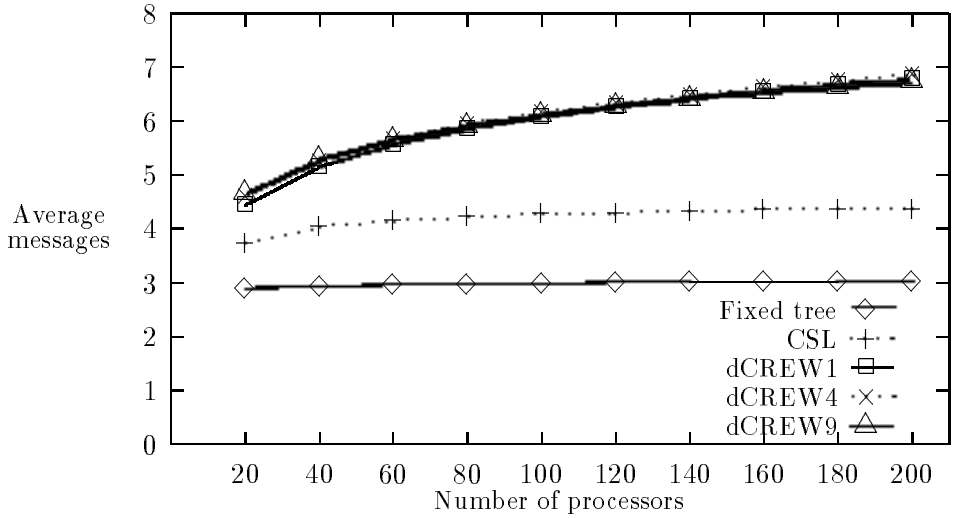


Figure 8: Average messages per CS entry with 200% load

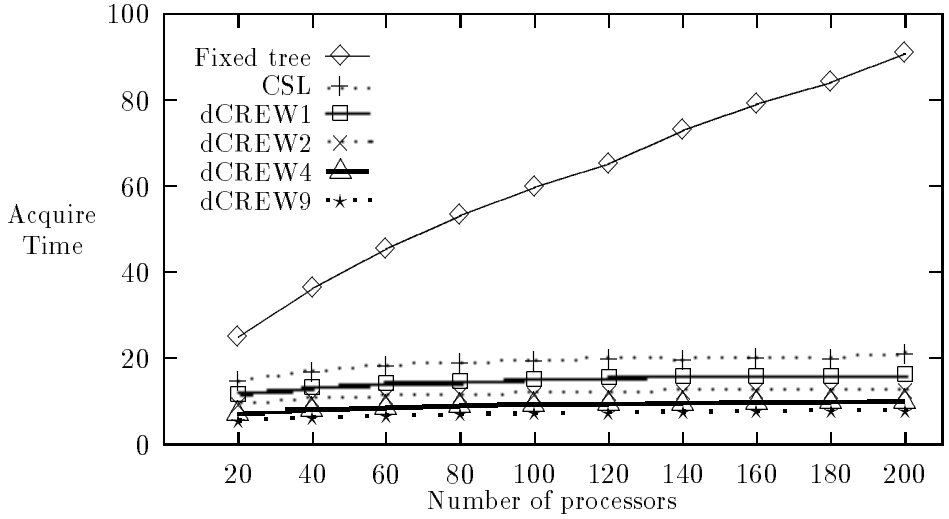


Figure 9: Token acquisition time with 50% load

The dCREW algorithm has a significantly lower acquisition time than the FT algorithm under all conditions tested, and a significantly lower acquisition time than the CSL algorithm under a high load. This is largely due to contention for the lock. In the mutual exclusion algorithms, a requesting process usually must join the waiting list, while the lock utilization is lower when concurrent readers are supported. Figure 11 most clearly shows this phenomena. The dCREW algorithm takes advantage of the higher proportion of readers to writers in the request stream, permitting a more highly parallel access to the resource, lowering token acquisition times.

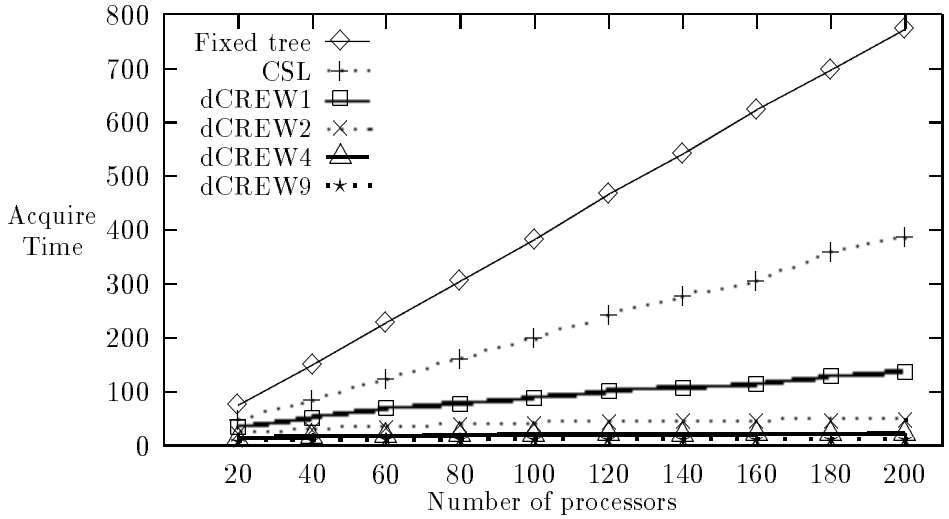


Figure 10: Token acquisition time with 100% load

### 5.3 Lock Utilization

The reduced lock acquisition time of the dCREW algorithm is due to the parallel access to the critical section. The parallelism should be reflected in a lower lock utilization. Let $T_{rhlock}$ be the sum of the times that the lock is being requested or in use, and $T_{sim}$ be the total simulation time. Then, the lock contention $C$ is defined as:

$$C = \frac{T_{rhlock}}{T_{sim}}$$

The lock contention under various loads is presented in Figures 12 through 14. Obviously, our algorithm shows less contention because more processors can be allowed to be in the CS at the same time. However, with higher loads, both algorithm expectedly approach to 100% contention as $T_{rhlock}$ increases nearly to $T_{sim}$. This is shown in Figure 14.

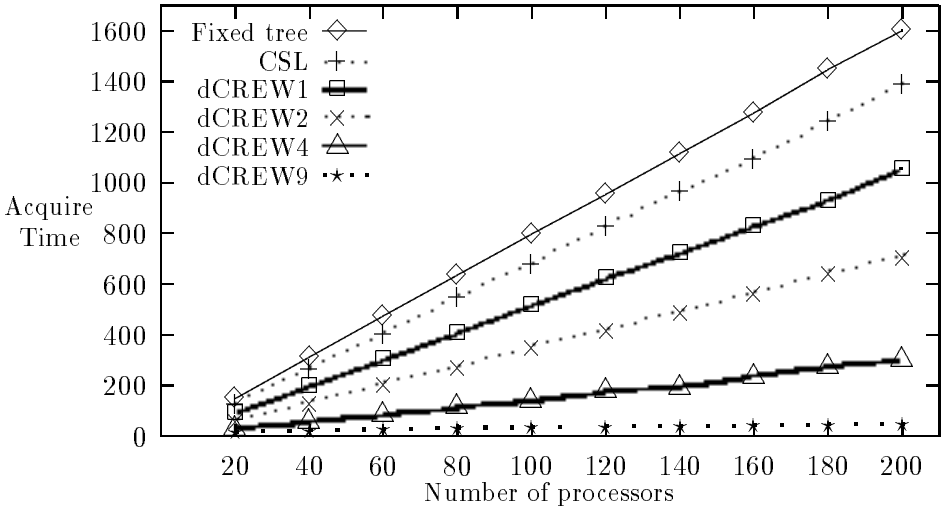


Figure 11: Token acquisition time with 200% load

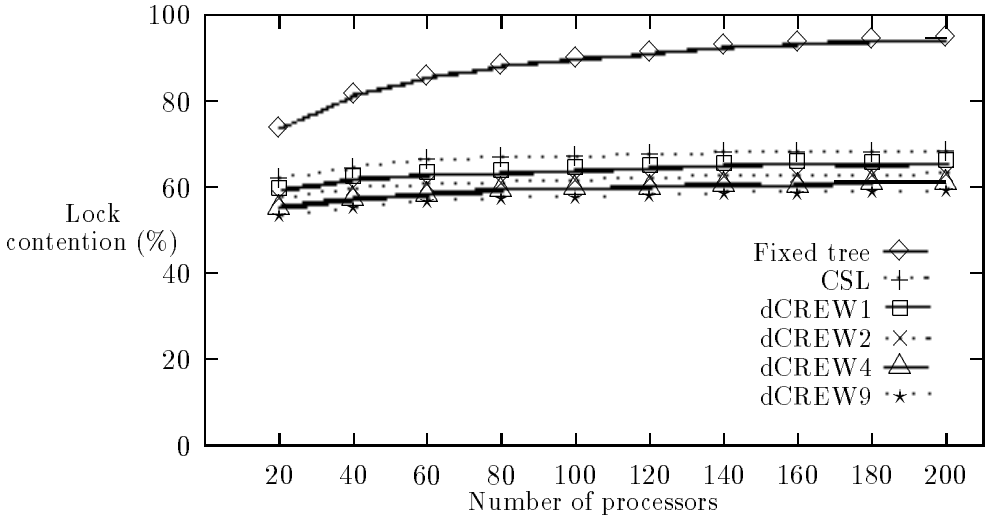


Figure 12: Lock contention with 50% load

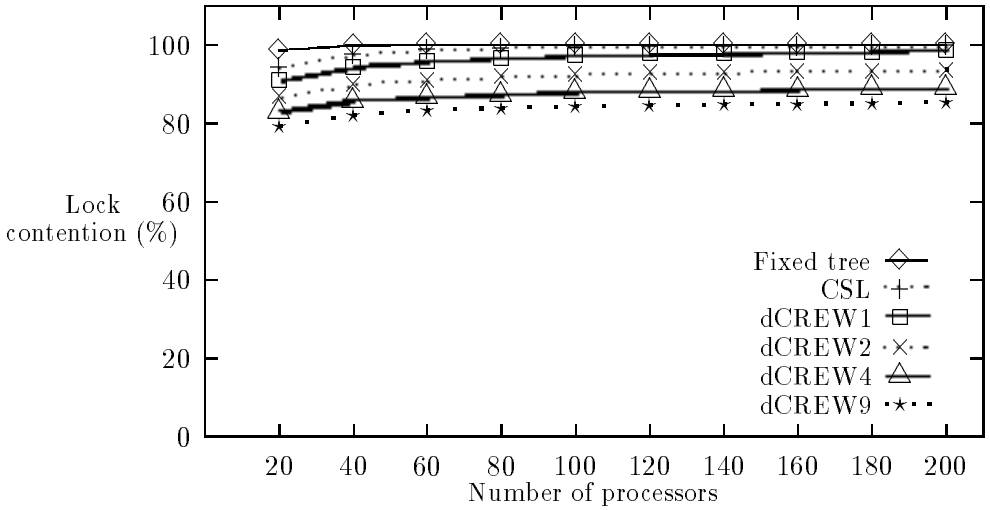


Figure 13: Lock contention with 100% load

# 6 Conclusion

We have presented an efficient, scalable algorithm for distributed concurrent-reader concurrent-writer synchronization. Our algorithm is based on the efficient path compression algorithm of Chang, Singhal, and Liu [3]. The $O(\log n)$ message passing overhead per request for CS operation and the $O(\log n)$ bits of storage overhead per processor make the algorithm scalable and practical for implementation. We evaluated the algorithm through experiments, and examined the performance against the FT and the CSL algorithms in terms of messages per CS entry, lock acquisition time, and lock utilization. The experiments were executed under varying system loads, varying ratios of readers to a writer, and varying numbers of processors. We found that the dCREW algorithm can significantly lower lock acquisition times and increase lock throughput at a small cost in an increased number of messages per critical section entry. Johnson [6] made a performance study of four $O(\log n)$ mutual exclusion algorithms [20, 21, 17, 3, 6], each of which is based on the fixed tree or the path compression approach, and found that the CSL algorithm had the best overall performance. This algorithm requires 5.6 messages per CS entry with 160 processors and a uniform load, while our algorithm requires about 6.5 messages exploiting more parallelism (Figure 8). Based on the algorithm that we present in this paper, one can easily extend it to give preference to readers or writers.

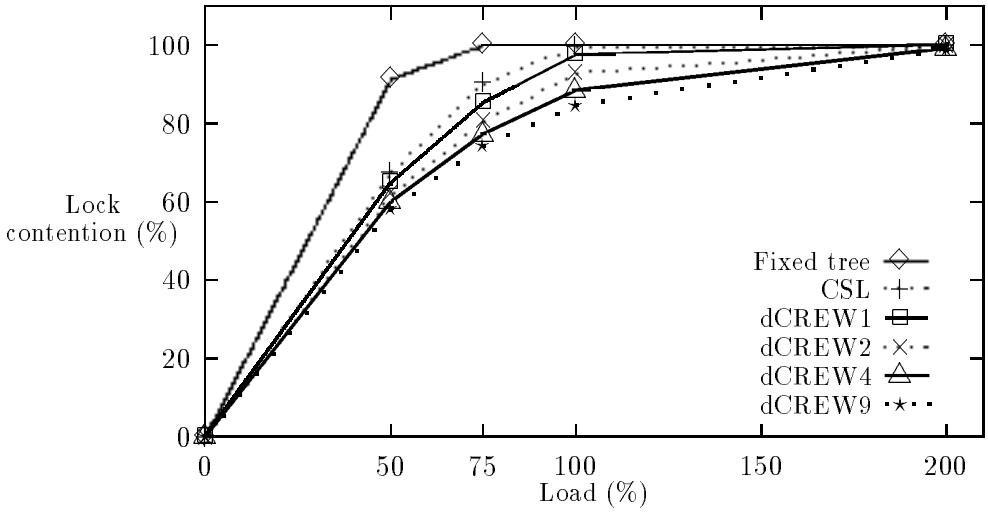


Figure 14: Lock contention when $n = 120$

## References


[1] O. S. F. Carvalho, G. Roucairol. On mutual exclusion in computer networks, technical correspondence. *Comm. of ACM*, 26(2):146–147, 1983.

[2] K. M. Chandy, L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Computer Systems*, 3(1):63-75, 1985.

[3] Y. I. Chang, M. Singhal, M. T. Liu. An improved $O(\log n)$ mutual exclusion algorithm for distributed systems. *Int'l Conf. on parallel Processing*, pages III295–302, 1990.

[4] D. Ginat, D. D. Sleator, R. Tarjan. A tight amortized bound for path reversal. *Information Processing Letters*, 31:3-5, 1989.

[5] Helary, Mostefaoui, and Raynal. An O(log n) Fault-Tolerant Distributed Mutual Exclusion Algorithm Based on Open-Cube Structure. *Intl. Conf. on Distributed Computing Systems*, pages 89-96, 1994.

[6] T. Johnson. A performance comparison of fast distributed mutual exclusion algorithms. *Proc. Int'l Parallel Processing*, pages 258-264, 1995.

[7] T. Johnson and R. Newman-Wolfe. A comparison of fast and low overhead distributed priority locks. *Journal of Parallel and Distributed Computing*, 1996.

[8] O. Krieger, M. Stumm, R. Unrau, and J. Hanna. A fair fast scalable reader-writer lock. *Proc. International Conference on Parallel Processing*, pages II201–204, 1993.

[9] A. Kumar. Hierarchical quorum consensus: A new algorithm for managing replicated data. *IEEE Trans. on Computers*, 40(9):994–1004, 1991.

[10] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, 1978.

[11] K. Li, P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. on Computer Systems*, 7(4):321–359, 1989.

[12] M. Maekawa. A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. on Computer Systems*, 3(2):145–159, 1985.

[13] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. *Proc. 3rd ACM Symp. on Principles and Practice of Parallel Programming*, Apr. 1991.

[14] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention, *Proc. Architectural Support for Programming Languages and Operating Systems*, pages 269-278, 1991.

[15] M. L. Neilsen, M. Mizuno. A DAG-based algorithm for distributed mutual exclusion. *Int'l Conf. on Distributed Computer Systems*, pages 354–360, 1991.

[16] M. Raynal. Distributed Algorithms and Protocols. John Wiley, 1988.

[17] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. on Computer Systems*, 7(1):61–77, 1989.

[18] G. Ricart, A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Comm. of the ACM*, 24(1):9–17, 1981.

[19] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Systems*, 4(2):180–209, 1979.

[20] M. Trehel, M. Naimi. A distributed algorithm for mutual exclusion based on data structures and fault-tolerance. *IEEE Phoenix Conference on Computers and Communications*, pages 36–39, 1987.

[21] M. Trehel, M. Naimi. An improvement of the $\log n$ distributed algorithm for mutual exclusion. *Proc. IEEE Int'l Conf. on Distributed Computer Systems*, pages 371–375, 1987.

[22] T. Johnson. Approximate Analysis of Reader/Writer Queues. *IEEE Transactions on Software Engineering*, 21(3):209-218, 1995.