Token-based Read/Write-Locks for Distributed Mutual Exclusion

Claus Wagner[1] and Frank Mueller[2]

[1] Computer Science Department, TECHNION, Haifa, Israel 32000
[2] Humboldt University Berlin, Institut f. Informatik, 10099 Berlin, Germany
mueller@informatik.hu-berlin.de, phone: (+49) (30) 2093-3011, fax: -3010

Abstract. The contributions of this paper are twofold. First, a protocol for distributed mutual exclusion is introduced using a token-based decentralized approach, which allows either multiple concurrent readers or a single writer to enter their critical sections. This protocol utilizes a dynamic structure incorporating path compression to keep the messages overhead low resulting in an average complexity of O(log n) messages per request. Second, this protocol is evaluated in comparison with another protocol that uses a static structure instead of dynamic path compression. The measurements show that although concurrent readers may require at most one additional message per entry, the concurrent execution of critical sections results in faster responses of up to 30% for short critical sections. For longer critical sections, savings in the overall execution time increase with the fraction of readers to up to 50%. In particular applications with large fractions of readers, *e.g.*, database queries, may exploit these benefits. The results further indicate that problems with fine-grained parallelism are more suitable for the dynamic protocol proposed here while the static protocol used for comparison performs equally well for coarse-grained parallelism. Overall, reader/writer distinction provides promising benefits in both cases.

1 Introduction

Common resources in a distributed environment may require that they are used in mutual exclusion. This problem is similar to mutual exclusion in a shared-memory environment. However, while the shared memory architectures generally provide atomic instructions (*e.g.*, test-and-set) to implement mutual exclusion, such provisions do not exist in a distributed environment. Furthermore, commonly known mutual exclusion protocols for shared-memory environments that do not rely on hardware support still require access to shared variables.

In distributed environments, mutual exclusion is provided via a series of messages passed between nodes that are interested in a certain resource. Several protocols to solve mutual exclusion for distributed systems have been developed [1]. They can be distinguished by their approaches as token-based and non-token-based. The former ones may be based on broadcast protocols or they may use logical structures with point-to-point communication.

In this work, we assume a fully connected network (complete graph). Network topologies that are not fully connected can still use our protocol but will have to pay additional overhead when messages from A to B have to be relayed via intermediate nodes. Second, we assume reliable message passing (no loss, duplications, or modifications of messages) but we do allow out-of-order message delivery with respect to a pair of nodes, *i.e.*, if two messages are sent from node A to node B, then they may arrive at B in a different order than they were sent. Our assumption is that local operations are several orders of a magnitude faster than message delivery since this is the case in today's networks and the gap between processor speed and network speed still seems to widen. Thus, our main concern is to reduce the amount of messages at the expense of local data structures and local operations.

We introduce a new protocol to provide mutual exclusion in a distributed environment that distinguishes read and write requests. The protocol is based on some ideas by Naimi *et al.* [10], *i.e.*, a decentralized protocol that did not distinguish readers and writers. We explain the design of our approach, illustrate it by examples and provide the pseudocode for it. Measurements in a simulation testbed are obtained to evaluate the performance of our protocol.

2 Related Work

A number of protocols exist to solve the problem of mutual exclusion in a distributed environment. Chang [1] and Johnson [4] give a more detailed overview and compare the performance of such protocols. Non-token based protocols establish consensus between nodes before entry to a critical section is granted and often employ global logical clocks and timestamps [6]. Token-based protocols link permission of entry to token ownership. The broadcast-based subclass of protocols does not require specific communication paths since all nodes always receive a message. Token-based protocols with logical structures constrain messages to certain paths, which may be static or they may change dynamically. Our work focuses on this last class of protocols since their message complexity of O(log n) is lower than that of most of the above approaches, which require at least O(n) messages on the average for a request in a network of n nodes. Exceptions are Maekawa with $O(\sqrt{n})$ realized by an evenly distributed set of nodes over their quorum (subset of nodes that have to grant access to a requester) [8] and Kumar with $O(n^{0.63})$ building on the former approach [5]. Raymond proposed an O(n) quorum protocol [12], which was improved by Srimani and Reddy [14]. However, quorum protocols with k concurrent entries to the critical section do not provide the distinction between an arbitrary number of readers and writers of our work.

Another protocol by Raymond [13] utilizes a static tree for request propagation and utilizes local queues for requests. This work was extended by Neilsen and Mizuno [11] to replace these local queues by a distributed queue. In contrast, Chang, Singhal and Liu [2] use a dynamic tree similar to Naimi *et al.* [10]. The protocol makes use of path compression similar to independent work on memory consistency protocols [7]. Our work was influenced by the approaches of Raymond and Naimi *et al.*. Our protocols are based on the same assumptions and use similar data structures for write requests, which also ensures that the complexity of the original protocols is preserved in our solution in the absence of read requests. The additional support of concurrent readers in our protocols then only adds a constant overhead of one message for a read chain, if this chains build up. Hence, the order of complexity of our protocols in terms of the number of messages remains the same as within the original protocols.

3 Dynamic Reader/Writer Protocol for Mutual Exclusion

The dynamic protocol utilizes a directed tree-like structure. The edges form a chain leading new requests from node R to the last requester L (or the token holder if no requests were pending). While a request is in transit, intermediate nodes set their edges to point to the new requester R, thereby providing path compression, *i.e.*, future requests will propagate directly to R from any of the intermediate nodes similar to Naimi *et al.* [10].

An example is depicted in Fig. 1 where a (read) request from A is sent via B to T. The intermediate nodes B and T have edges directed at A afterwards. If T is still engaged in the critical section with a write lock, the read request cannot be served yet. Instead, T creates a `next` pointer (dashed edge) to A indicating the next recipient of the token. Once T exits its critical section, it sends the token to A and removes the `next` edge. A proceeds with its critical section under read protection. At this time, C issues a read request that is sent to A via B resulting in new edges from A and B to C. A responds by sending the token to C even before exiting its critical section because both A and C may execute concurrently in their critical sections under read protection. In addition, A registers C as the next reader (dotted edge) and C notes the fact that it is the last reader in a chain of readers. Node C will hold on to the token until (1) a request from another reader arrives next or (2) C and A exit their critical sections. The former case allows more concurrent readers to be served if they arrive ahead of writers. The latter case ensures that a writer will only be served after all reads have completed. Once A exits its critical section, it sends an acknowledgement to C, which must be received by C before the token may be forwarded to a writer.

The example illustrates that the protocol requires directed edges that requests travel along. It includes a distributed next queue of pending requests whose source is the token holder while the sink is the last requester. A distributed read queue links concurrent readers starting at the earliest requester still engaged in a critical section via consecutive requesters to the sink of the reader chain. Consecutive members of the read chain are either in the critical section or have not received an acknowledgement from their predecessor in the read chain yet. When a node issues multiple read requests in a row, the read chain may be circular imposing the necessity to log pending recipients of acknowledgements in a local FIFO queue. Requests are always handled in the order that they arrive at the tail of the request queue, *i.e.*, a write request always terminates a

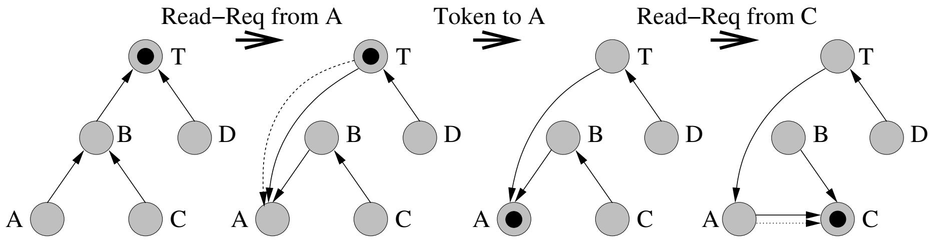

Fig. 1. Simple Example

chain of readers and subsequent read requests are served after the write. This ensures fairness and avoids starvation. The following notation is used in order to reflect these requirements when describing the protocol:

Token is true if nobody is engaged within a critical section.

Dir represents an edge that points towards the last requester or, in the absence of requests, to the token holder, creating a distributed tree (solid edges).

Next points to the next requester of the token. The next chain represents a distributed queue of pending requests (dashed edges).

Next_readers represents a FIFO list of readers with outstanding acknowledgements. The next readers over all nodes form a chain of concurrent readers that will be reduced successively by acknowledgements (dotted edges).

Pending_Acks counts the number of acknowledgements a node still expects. When positive, a node may only forward the token, if it possesses it, to a reader since other readers are still active. Only when the value because zero, may the token be forwarded to a writer.

Token_mode is write if a writer is engaged in a critical section. The value is read if the last reader (of the reader chain) is in a critical section or multiread if the token has already been sent to a concurrent reader node. It is undef in any other case.

Next_mode is read/write if next points to the next R/W requester, respectively.

The pseudocode of the protocol is depicted in Fig. 2. Write requests are handled similar to the protocol by Naimi *et al.* [10]. Next, the relation between the abstract operational model and the actual implementation shall be discussed.

At initialization, all nodes point via dir to the start node, which is the token holder. Notice that the edge of the token holder to himself is omitted in the example since it represents a special case beyond the tree structure. A lock request for a locally unused token can be served right away. All other requests propagate along the dir edges while the requester clears his dir pointer since he is the last requester waiting for the token to arrive. Once the token arrives, the flag to indicate if the node should "expect an acknowledgement" is set according to the value piggybacked with the token message and the number of pending acknowledgements may be incremented accordingly. In case of concurrent reads, the token is forwarded to the next reader and the mode is set to multiread. Otherwise, the request mode is stored before entering the critical section. The token flag is also cleared during the critical section indicating that it is in use.

Upon the end of the critical section (unlock), several cases are distinguished. Readers in the chain (except for the tail of the chain) send an acknowledgement to the next member of the chain, thereby reducing the chain if they have already received an acknowledgement from their predecessor. If a next requester exists and all read requests have completed (no pending acknowledgements), the token is sent to the next requester. Otherwise, the token is marked as locally available unless it was already sent to a concurrent reader at an earlier time.

A receiver of a request from some sender also has to distinguish certain cases before changing its dir edge to the sender. If the receiver is not the last requester, then he simply forwards the request with the sender's id along the directed edges.

```
token = (self == start); // TRUE if token locally available
dir = start;             // edge with changing destination for requests propagation
next = NULL;             // next receiver of token (⇒ dist. Q of unserved requesters)
next_readers = φ;        // FIFO list of next readers (⇒ dist. Q of conc. readers)
pending_acks = 0;        // number of outstanding acknowledgements
token_mode = undef;      // protection of critical section (undef/read/multiread/write)
next_mode = undef;       // protection for next receiver (undef/read/write)
```

```
PROC lock(mode) IS
    IF ¬ token THEN
        SEND request(self, mode) to dir;
        dir = self;
        AWAIT(token(&expect_ack));
        IF expect_ack THEN
            pending_acks += 1;
        ENDIF;
        concurrent_read = (mode == read
            AND next_mode == read);
        IF next ≠ NULL AND
            concurrent_read THEN
            SEND token(TRUE) to next;
            append(next_readers, next);
            next = NULL;
            token_mode = multiread;
        ELSE
            token_mode = mode;
        ENDIF;
    ELSE
        token_mode = mode;
    ENDIF;
    token = FALSE;
END lock;

PROC unlock IS
    IF token_mode == multiread AND
        pending_acks == 0 THEN
        SEND ack to head(next_readers);
        delete_head(next_readers);
    ENDIF;
    IF next ≠ NULL AND
        pending_acks == 0 THEN
        SEND token(FALSE) to next;
        next = NULL;
    ELSE IF token_mode ≠ multiread THEN
        token = TRUE;
    ENDIF;
    token_mode = undef;
END unlock;
```

```
PROC receive_request(sender, mode) IS
    IF dir ≠ self THEN
        SEND request(sender, mode) to dir;
    ELSE
        concurrent_read = (token_mode ==
            read AND mode == read);
        IF token AND pending_acks == 0
            OR concurrent_read THEN
            SEND token(¬ token) to sender;
            token = FALSE;
            IF concurrent_read THEN
                token_mode = multiread;
                append(next_readers, sender);
            ENDIF;
        ELSE
            next = sender;
            token = FALSE;
            next_mode = mode;
        ENDIF;
    ENDIF;
    dir = sender;
END receive_request;

PROC receive_ack IS
    pending_acks -= 1;
    IF pending_acks > 0 OR
        ¬ empty(next_readers) AND
        token_mode == undef THEN
        send ack to head(next_readers);
        delete_head(next_readers);
    ELSE IF token AND pending_acks == 0
        AND next ≠ NULL THEN
        send token(FALSE) to next;
        next = NULL;
        token = FALSE;
    ENDIF;
END receive_ack;
```

Fig. 2. Pseudocode of Read/Write-Lock Protocol

Otherwise, one of two cases may apply. If the token is available and all readers have completed (no acks pending) or if the request is for a concurrent read, the receiver sends the token to the sender, clears the token status and records the next concurrent reader if necessary. The piggybacked flag is true if the new request represent a concurrent read. If neither the token was available nor was the request for a concurrent read, the sender is logged as the next requester.

Upon reception of an acknowledgement, the number of pending acknowledgments is decremented and an acknowledgement is sent to the next reader if this had not already been done in the unlock operation. Otherwise, the token is forwarded to the next requester if all acknowledgements have been sent and the read chain has collapsed. Notice that the next requester was either a writer or a reader at the head of another read chain so that the piggybacked value FALSE requires no checks for acknowledgments by this requester (similar as in unlock).

The protocol ensures mutual exclusion since either (a) a single writer may own the token or (b) the tail of the read chain owns the token (and predecessors in the read chain may execute their critical read section concurrently). Deadlocks are avoided since requests are forwarded via a tree-like structure to the tail of the requester queue. These requests are then served in the order that they were queued according to the mutual exclusion property, *i.e.*, circular dependencies cannot occur. Starvation is avoided by the FIFO policy of queuing requests, *i.e.*, a new request will be served exactly after all its predecessors in the FIFO queue have been served (again subject to mutual exclusion properties for readers/writers). The protocol has a complexity of O(log n) messages since it uses the same tree structure and messages for requests as Naimi *et al.* [10] although a constant of one message per reader in a chain is added.

4 Experimental Platform

The experiments were conducted in a simulation environment where distributed nodes were mapped onto processes and communication was performed via asynchronous TCP operations on a uniprocessor. Each process is itself multi-threaded to allow asynchronous communication. Two components implemented as threads require further explanation: A user thread may execute application code including lock and unlock operations. The communication server receives requests, tokens and acknowledgements. Requests are serviced right away while the other messages are registered. If a user thread awaits such a message, he is activated. Otherwise, the message is stored and when the user thread performs the next await operation, this operation will not block. These operations are implemented using a POSIX mutex and condition variables. In addition, each operation of the protocol depicted in Fig. 2 is executed within mutual exclusion, *i.e.*, the protocol ensures a monitor semantics to keep its internal state consistent, which arbitrates the threads on a node. The monitor is realized by the POSIX mutex also used in conjunction with the condition variables. In prior experiments, the testbed has proved its adequacy to evaluate different mutual exclusion protocols yielding results comparable to prior work [3].

The main experiment consists of a synthetic benchmark as a user thread along the lines of prior work [3, 9]. A loop of i iterations consists of a critical section followed by non-critical code. Durations of the critical section ξ and of non-critical code Γ can be varied to simulate different request rates. Requests are randomized by $\pm 10\%$ around ξ and Γ providing a framework of contention where requests are clustered around a certain point in time but arrive in random orders, as previously shown in [15]. The number of distributed nodes N can be varied and the fraction of readers may be specified to evaluate the protocol.

This setup allows the measurement of various aspects. First, the number of messages exchanged per entry (NME) of a critical section can be determined. This allows a comparison between different methods and parameters to relate timing results to message overhead. Second, the average response time for a series of requests is calculated. This provides the means to determine savings per critical section entry. Third, the overall execution time of a test run may be measured. This indicates the overall savings that may be obtained in an application. Varying the proportion of readers and writers gives an insight on how a protocol performs for a varying number of read requests. But it also provides a comparison with the base protocol without distinction between readers and writers, namely when the fraction of read locks is set to 0.

The testbed includes the implementation of the dynamic tree-structured protocol for token-based mutual exclusion in a distributed environment, called DVARW in the following. In addition, a token protocol based on a static tree structure has also been implemented. The base protocol by Neilsen and Mizuno [11] has been extended to distinguish read and write locks [15] and is referred to as SVARW in the following. Neilsen and Mizuno allow requests only to travel along the edges of the static tree. An edge may reverse its direction. Thereby, edges form a chain pointing to the last requester. In contrast, Naimi's approach adopted in this work modifies the sink of edges, not just their direction, as requests propagate along paths. This results in path compression intended to reduce the number of hops for future requests. The modifications to Neilsen and Mizuno's algorithm to distinguish readers and writers are similar to the protocol described in this paper including token forwarding for concurrent reads before completing a critical section and sending acknowledgements along the chain of concurrent readers upon completion of the critical section. The details of the static reader/writer protocol are beyond the scope of this paper (see [15]).

5 Measurements

The measurements were obtained on a 133MHz Pentium under Linux 2.1 executing the benchmark described in the aforementioned testbed. The number of iterations was $i = 30$. The duration of a critical section was chosen as $\xi = 200\mu sec$ while the non-critical part was $\Gamma = 2000\mu sec$. This choice reflects that critical sections tend to be short while non-critical code takes up more of the overall execution time. The choice of values was also influenced by prior simulation studies on the performance of distributed mutual exclusion protocols [3, 9].

Fig. 3 depicts the results for the dynamic tree protocol under varying number of nodes ($N = 2..64$) and varying fractions of read requests. Fig. 3(a) shows that the number of messages depends on the total number of nodes in the system, which is caused by an increasing path length to propagate requests. For a given

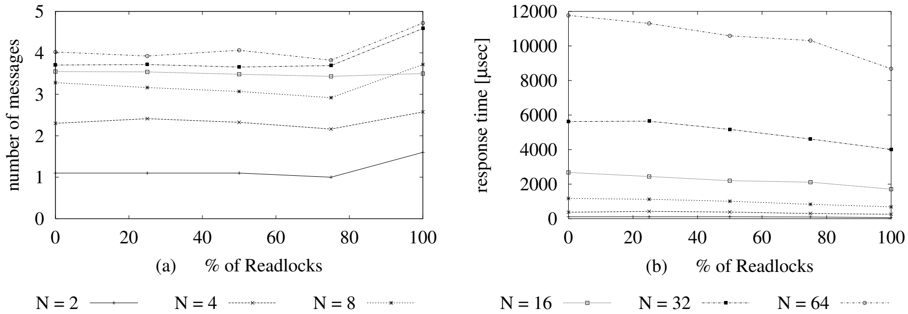

Fig. 3. Messages and Response Time of the Dynamic Tree Protocol (DVARW)

number of nodes, the number of messages remains constant up to a certain point. This indicates that the dynamic protocol with read/write distinction performs just as well as the base protocol without this distinction, which underlines the qualities of our approach. At some point, read requests dominate (>75%) and the number of messages increases by almost 1 at 100%. This effect is caused by acknowledgement messages that are only sent when read chains start to build up at high reader frequencies. Fig. 3(b) indicates that the response times increases linearly with the number of nodes. Furthermore, with an increasing portion of read requests, the response time is reduced up to 30% for $N = 64$. The actual reduction (the gradient of each curve) increases with the number of nodes. This demonstrates that although concurrent readers may require an additional message, the concurrent execution of critical sections results in faster responses. In other words, applications can benefit from read concurrency with this protocol (quantified more precisely in Fig. 5).

In contrast, Fig. 4 depicts the results for the static tree protocol. The number of messages remains unchanged or increases slightly, depending on the number of nodes, since messages have to propagate along longer paths within the static tree. The response time also increases linearly for the number of nodes while it decreases with higher reader frequencies. While the response time of the static variant is higher than in the dynamic counterpart in the absence of readers, the results are about the same in the absence of writers. In the latter case, the number of messages is also about the same, which explains this behavior.

Fig. 5 summarizes the impact of a read/write distinction for an entire execution of the benchmark. Here, the number of iterations was $i = 20$, there

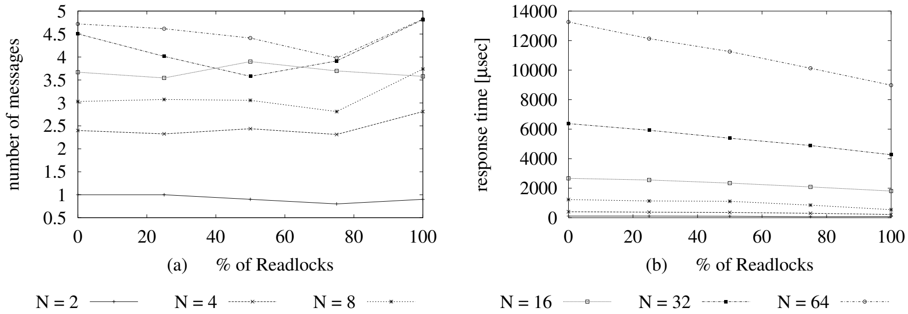

Fig. 4. Messages and Response Time of the Static Tree Protocol (SVARW)

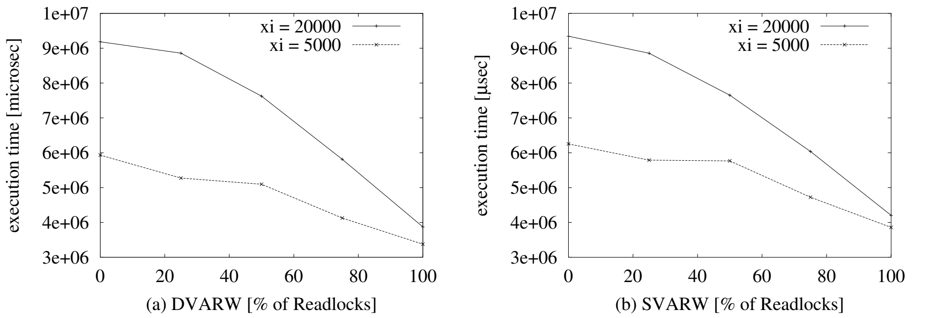

Fig. 5. Overall Execution Time (Dynamic vs. Static Tree)

were $N = 16$ nodes, the critical section was $\xi = 5000\mu sec$ and $\xi = 20.000\mu sec$ while non-critical code took $\Gamma = 2000\mu sec$. This simulates a case where complex operations in critical sections dominate non-critical code. Both the dynamic protocol in Fig. 5(a) and the static protocol in Fig. 5(b) show execution time savings of around 50% when all requests are for read access compared to only write requests. The savings are slightly higher for longer critical sections than for shorter ones. While some savings already materialize around 50% readers, the savings increase even more rapidly for a higher proportions of readers. Typically, applications experience far more read than write requests, *e.g.*, database queries. Hence, these savings have an impact on the overall performance.

The last experiment was conducted to determine the impact of different request frequencies on the protocol. For this purpose, Γ was varied for $N = 32$ nodes, $i = 20$ iterations and $\xi = 200\mu sec$ per critical section. Figure 6 shows that for high request frequencies (small Γ) the dynamic protocol results in fewer messages (about one messaged saved) and shorter response times (by about 10%) compared to the static case. For low request frequencies, the differences between the protocols diminishes. This is caused by the properties of path compression in the dynamic protocol where the structure of the dynamic tree degenerates into a list under these circumstances. This shows that the dynamic protocol exhibits its full advantage mostly for applications which use synchronization heavily throughout its execution. In short, problems with fine-grained parallelism are more suitable for the dynamic protocol while the static protocol performs equally well for coarse-grained parallelism. Overall, reader/writer distinction provides promising benefits in both cases.

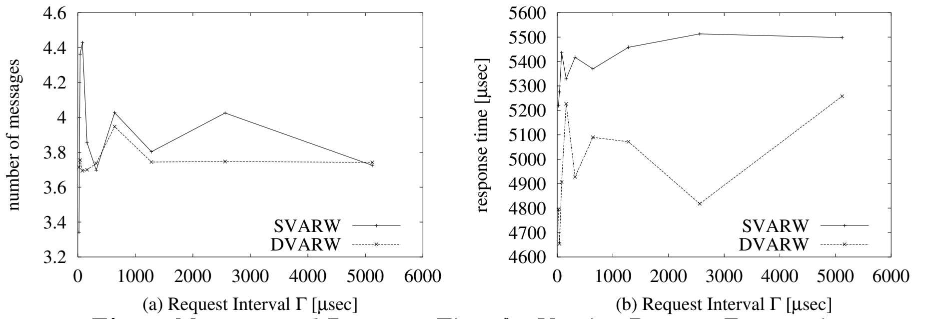

Fig. 6. Messages and Response Time for Varying Request Frequencies

6 Conclusions

We developed a protocol for distributed mutual exclusion that distinguishes reader and writer requests utilizing a dynamic tree structure to reduce message overhead resulting in an average complexity of O(log n) messages per request. This protocol was evaluated in comparison with a protocol using a static structure. The measurements show that although concurrent readers may require at most one additional message per entry, the concurrent execution of critical sections results in faster responses of up to 30% for 64 nodes when critical sections are short. For longer critical sections, savings in the overall execution time increase with the fraction of readers to up to 50%. In particular applications with large fractions of readers, *e.g.*, database queries, may exploit these benefits. The results further indicate that problems with fine-grained parallelism are more suitable for the dynamic protocol proposed here while the static protocol used for comparison performs equally well for coarse-grained parallelism. Overall, reader/writer distinction provides promising benefits in both cases.

References

1. Y. Chang. A simulation study on distributed mutual exclusion. *J. Parallel Distrib. Comput.*, 33(2):107–121, March 1996.
2. Y. Chang, M. Singhal, and M. Liu. An improved O(log(n)) mutual exclusion algorithm for distributed processing. In *Int. Conference on Parallel Processing*, volume 3, pages 295–302, 1990.
3. S. Fu, N. Tzeng, and Z. Li. Empirical evaluation of distributed mutual exclusion algorithms. In *International Parallel Processing Symposium*, pages 255–259, 1997.
4. T. Johnson. A performance comparison of fast distributed mutual exclusion algorithms. In *Proc. 1995 Int. Conf. on Parallel Processing*, pages 258–264, 1995.
5. A. Kumar. Hierachical quorum consensus: A new algorithm for managing replicated data. *IEEE Trans. On Computers*, 40(9):994–1004, 1991.
6. L. Lamport. Time, clocks and ordering of events in distributed systems. *Comm. ACM*, 21(7):558–565, June 1978.
7. K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Systems*, 7(4):321–359, November 1989.
8. M. Maekawa. A sqrt(n) algorithm for mutual exclusion in decentralized systems. *ACM Trans. on Computer Systems*, 3(2):145–159, 1985.
9. F. Mueller. Prioritized token-based mutual exclusion for distributed systems. In *International Parallel Processing Symposium*, pages 791–795, 1998.
10. M. Naimi, M. Trehel, and A. Arnold. A log(N) distributed mutual exclusion algorithm based on path reversal. *J. Parallel Distrib. Comput.*, 34(1):1–13, April 1996.
11. M. L. Neilsen and M. Mizuno. A dag-based algorithm for mutual exclusion. In *Distributed Computer Systems*, pages 354–360, 1991.
12. K. Raymond. A distributed algorithm for multiple entries to a critical section. *Information Processing Letters*, 30(4):189–193, February 1989.
13. K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Systems*, 7(1):61–77, February 1989.
14. P. Srimani and R. Reddy. Another distributed algorithm for multiple entries to a critical section. *Information Processing Letters*, 41(1):51–57, January 1992.
15. C. Wagner. Algorithmen zum gegenseiten Ausschluß in verteilten Systemen. Master's thesis, Humboldt University Berlin, Germany, December 1999.