Prioritized Token-Based Mutual Exclusion for Distributed Systems

Frank Mueller
Humboldt-Universität zu Berlin, Institut für Informatik, 10099 Berlin (Germany)
e-mail: mueller@informatik.hu-berlin.de phone: (+49) (30) 20181-276 fax:-280

Abstract

A number of solutions have been proposed for the problem of mutual exclusion in distributed systems. Some of these approaches have since been extended to a prioritized environment suitable for real-time applications but impose a higher message passing overhead than our approach. We present a new protocol for prioritized mutual exclusion in a distributed environment. Our approach uses a token-based model working on a logical tree structure, which is dynamically modified. In addition, we utilize a set of local queues whose union would resemble a single global queue. Furthermore, our algorithm is designed for out-of-order message delivery, handles messages asynchronously and supports multiple requests from one node for multi-threaded nodes. The prioritized algorithm has an average overhead of $O(log(n))$ messages per request for mutual exclusion with a worst-case overhead of $O(n)$, where n represents the number of nodes in the system. Thus, our prioritized algorithm matches the message complexity of the best non-prioritized algorithms while previous prioritized algorithms have a higher message complexity, to our knowledge. Our concept of local queues can be incorporated into arbitrary token-based protocols with or without priority support to reduce the amount of messages. Performance results indicate that the additional functionality of our algorithm comes at the cost of 30% longer response times within our test environment for distributed execution when compared with an unprioritized algorithm. This result suggests that the algorithm should be used when strict priority ordering is required.

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) that can be exploited to provide mutual exclusion, such provisions do not exist in a distributed environment. Furthermore, commonly known mutual exclusion algorithms 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 algorithms to solve mutual exclusion for distributed systems have been developed [2]. 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.

We introduce a new algorithm to provide mutual exclusion in a distributed environment that supports priority queuing. The algorithm uses a token-passing approach with point-to-point communication along a logical structure. Each node keeps a local queue and records the time of requests locally. These queues form a virtual global queue ordered by priority and FIFO within each priority level with regard to the property of relative fairness [14]. We neither use a global logical clock nor timestamps to avoid the associated message-passing overhead.

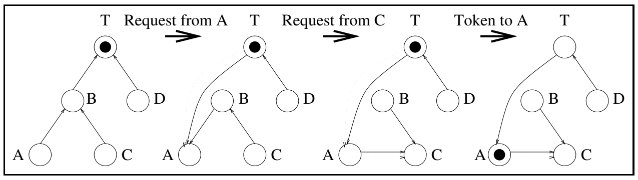


Figure 1. Unprioritized Example

The main purpose of this algorithm is to provide an efficient solution to the problem of mutual exclusion in distributed systems with known average and worst-case behavior and to support strict priority scheduling for real-time systems at the same time. Thus, the worst-case behavior of the algorithm can be determined for known bounds on message delays. We are also aiming at enhancing distributed programming environments by priority support in a POSIX-like manner [20], *e.g.* within our DSM-Threads environment [11, 12]. The resulting algorithm works asynchronously in out-of-order message systems while we assumed ordered message delivery in our previous work [13].

In the following, the algorithm is derived from a non-prioritized algorithm that uses a single distributed queue [16]. We explain why this algorithm is not easily extended to handle priorities. Instead, we develop our algorithm step-by-step by introducing priority support, local queues and token forwarding. Then, we present performance results. [14] shows the correctness of the algorithm.

1.1 The Model

First, 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 that they were sent from A. 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. For the property of *relative fairness* for priority request (discussed in [14]), we also assume bound message delays, *i.e.* deterministic latency and throughput characteristics of the communication medium.

1.2 The Unprioritized Algorithm

The basic idea behind our token-passing protocol for distributed mutual exclusion stems from an algorithm by Naimi *et. al.* [16]. In this decentralized algorithm, nodes form a logical tree pointing via *probable owners* towards the root. Consider Figure 1. The root T holds the token for mutual exclusion. A request for mutual exclusion travels along the probable owners (solid arcs) to

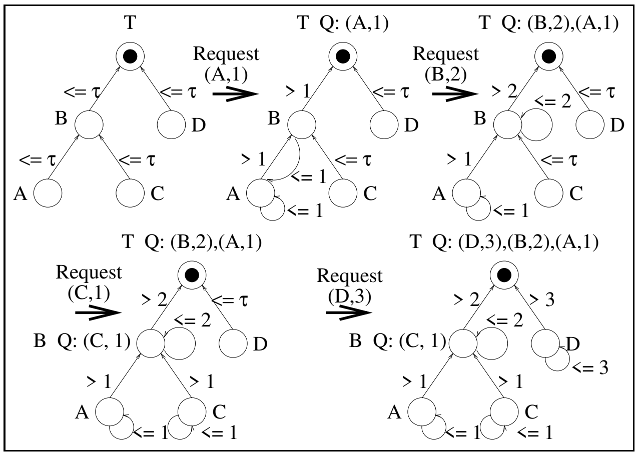


Figure 2. Example of Token Requests

the token holder. In the example, the request by A is send to B. Node B forwards the request along the chain of probable owners to T and sets its probable owner to A. When a request arrives at the current token holder, the probable owner and a *next* pointer are set to the requester, *i.e.* T sets the *next* pointer (dotted arc) and the probable owner to A. The next request from C is first sent to B. Node B forwards the request to A following the probable owners and sets its probable owner to C. Node A sets its *next* pointer and probable owner to C. When the token holder returns from its critical section, the token is sent to the node that *next* points to. In the example, T passes the token to A and deletes the *next* pointer.

Consider the process of request forwarding again. When a request passes intermediate nodes, the probable owner is set to the requesting node. This causes a transformation of the tree to a forest. The traversed nodes form a tree rooted in the requesting node, which will be the token holder of the future. Nodes that have not been traversed are still part of the original tree rooted in the current token holder. For example, when B forwarded A's request to T, B sets the probable owner to A. At this point, one logical tree $C \rightarrow B \rightarrow A$ is rooted in A while another tree $D \rightarrow T$ is still rooted in T. Once the request reaches T, the separate trees are merged again (2nd tree of Figure 1).

Should k requests be in transit, then there may be up to $k + 1$ separate trees in the forest. Once all requests have arrived at current or future token holders, the forest collapses to a single tree again rooted in the last requester. The *next* pointers form a distributed queue of pending requests from the current token holder to the last requester. A new request would simply be appended at the end as described above.

This algorithm has an average message overhead of $O(log(n))$ since requests are propagated through a tree. In the worst case, requests can be forwarded via $n - 1$ messages and one additional message is needed to send the token. However, the model assumes that there is only one requester per node at a time and requests are ordered FIFO without priority support. In a multi-threaded environment, multiple requests may be issued from a node. The algorithm could be easily extended to support a list of *next* pointers per node, where each pointer represents a request of a different thread. However, priority queuing cannot be supported easily as discussed in the next section.

2 Priority Queuing Support

In a prioritized environment, requests should be ordered first by priority and then (within each priority level) FIFO. The basic idea for priority support is to accumulate priority information on intermediate nodes during request forwarding. When a request issued by node N at priority $\Im(N)$ passes along a node I, the probable owner for requests with priority $\leq \Im(N)$ is set to N. (Higher priorities denote more important requests.)

The example in Figure 2 depicts a sequence of token requests of A, B, C, and D with priority 1, 2, 1, and 3, respectively. (Disregard the local queues of nodes for now.) The token resides in node T while the requests arrive. For the sake of simplicity, we assume that the initial logical structure is a tree whose edges point to the token owner for all priorities (labeled as top τ). Each request results in new forwarding information for certain priority levels, as depicted by the labeled edges. For example, request $(A, 1)$ results in edges $A \rightarrow A$ labeled ≤ 1, which indicates that future requests from A with priority ≤ 1 are handled by A. Other edges include $A \rightarrow B(> 1)$, $B \rightarrow T(> 1)$ and $B \rightarrow A(\leq 1)$.

If we follow the unprioritized algorithm with the addition of priority constraints on edges, we had to set *next* pointers for each request. However, a *next* pointer may have to be modified when higher priority requests arrive before lower ones have been served. This indicates that the unprioritized algorithm is unsuitable to support priority queuing. In essence, it is not possible to insert new requests at arbitrary places within the distributed queue due to race conditions. If a new entry had to be added at the head of the queue and the request arrives at the current token holder, the *next* pointer of the requester R has to be set to the *next* pointer of the current token holder. Yet, while the request was in transit, other nodes may have already registered more requests with R, thereby utilizing its *next* pointer. Thus, there is a race between setting R's *next* pointer. This deficiency can be solved by local queues.

3 Local Queues

The unprioritized algorithm uses a distributed queue. We can replace this queue (*i.e.*, the *next* pointers) by a set of local queues. When a node R issues a request at priority $\Im(R)$, this request propagates along the chain of probable owners until it reaches a node I with a pending request at priority $\geq \Im(R)$. It is then locally queued at I. The basic idea is that node I has to be served before our request. Thus, R's request may be stored locally until the token arrives. Once the token arrives, R's request should be served after I's if no other requests have arrived since then. This is also depicted in Figure 2, where local queues are associated with nodes T and B. Node B stores C's request at priority one since B has an outstanding request at priority two that will be served before.

A subtable problem with a set of distributed queues is the handling of FIFO queuing at the same priority level. Consider T sending the token to I. Node T has an entry for node I at the head of its local queue and possibly more entries at the same or at a lower priority. When I receives the token, the queue of T is piggybacked. Node I then has to merge its local queue and T's queue. However, if two entries have the same priority in different queues, which entry should be served first?

In a distributed system, FIFO policies can be enforced by a global logical clock via timestamps [8, 4]. However, timestamps result in additional message-passing overhead for communication with processes that never request a certain lock and overhead for acknowledgments. Instead, we utilize *local time* facilities to enforce FIFO ordering. Notice that local time only has a meaning on the current node. However, when the token is transmitted from T to I and T's queue is piggybacked, then the first request in T's queue is a request by node I. If all requests by I log the local initiation time, we are able to measure the time between request initiation $t_I(req)$ and token reception on the local node (now). This interval is called the request latency $t_I(latency)$ (see Figure 3). The latency itself is composed of the request transit time $t_I(req\_trans)$, the token transit time $t_I(token\_trans)$, and the token usage time $t_I(usage)$.

$$t_I(latency) = t_I(req\_trans) + t_I(token\_trans) + t_I(usage)$$

Index I may be replaced by other indices for other nodes. Notice that the queue piggybacked with a token message contains local requests *and* the accumulated token usage time. The token usage time can be calculated as the time spent in critical sections on nodes holding the token since request reception. Thus, latency and

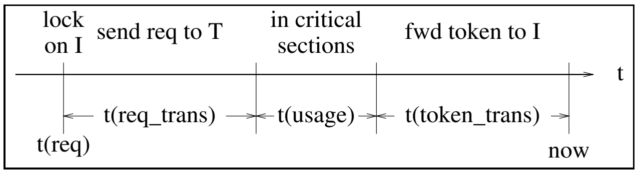


Figure 3. Events and Times for Token Requests

usage can be measured but the transit times may only be inferred indirectly due to a lack of global clocks.

We can now determine the approximate usage times for nodes X within the local queue due to the following observation. The local queue contains requests from the current node and possibly other nodes with the associated reception time $t_X(receipt)$ of each request by the local node. Thus, the usage time for local requests from X is determined relative to the request by I on the head of the piggybacked queue:

$$t_X(new\_usage) = t_X(usage) + (now - t_X(receipt)) * (t_I(usage)/t_I(latency))$$

The usage time on the current node for request X is estimated as the portion of time since receiving request X relative to I's latency. Consequently, X's local usage time is smaller than I's accumulated usage time, which is consistent since X was received after I had been issued. Once the usage times have been determined for local requests, the local queue can be merged with the piggybacked queue from T, which already contains accumulated usage times. Requests are ordered by descending priority and, within each priority level, by descending usage time. Thus, the usage time provides a means of aging that guarantees requests to be eventually served. This avoids starvation of requests, discussed in more detail in [14]. The reception time of all requests is set to the current local time after merging.

The token may be used to grant access in mutual exclusion on the local node. During token usage, other requests may be received, which are then logged with a zero usage time and the current local time as a reception time. These requests are merged into the existing queue using the same precedence rules as before (descending priority and descending usage time within each priority) and, in addition, with descending reception time (within the same priority level and the same usage times).

4 Token Forwarding

Once the token is no longer being used on the local node, it may be forwarded to the first requester within the queue. The queue is also piggybacked with the forwarded token but usage times are recalculated before the forwarding to include the local usage time:

$$t_X(new\_usage) = t_X(usage) + now - t_X(receipt)$$

The example of Figure 2 is continued in Figure 4. The token is forwarded to nodes D and B. The previous token holder retains only one link to the next token holder for requests at any priority level. Notice that the figure does not depict usage times and their adjustment upon token forwarding.

During request propagation within the logical tree, a request from node R propagates along intermediate nodes I before reaching the token holder T, as depicted in Figure 5a. Token forwarding may, however, result in a race condition that can be characterized as follows. A request from a node R may pass intermediate node I, where another lower priority request had already been issued. If the token is sent from node T while request R is in transit and has already passed node I, a race between the request and the token location exists. For example, in Figure 4 consider a request $(A, 4)$ being issued and passed *via* B and T. Meanwhile, T has already forwarded the token to D and D has forwarded it to B. Hence, the new request would trail behind to D and B, the latter being a node the request had passed once before (forming a cycle).

Returning to the abstract characterization of the race condition, the communication overhead may no longer be bound by the number of nodes in the system since the request from R may pass repeatedly by node I. We can avoid this situation by adding request

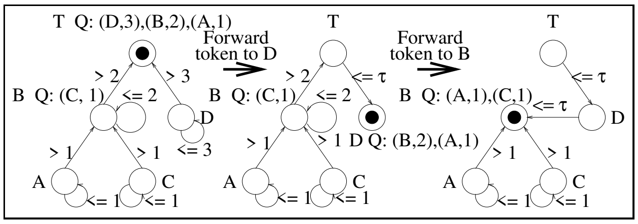


Figure 4. Example of Token Forwarding

R to the local queue of all intermediate nodes I. Should the token then be passed to I before request R reaches T (but after passing by I), node I would forward the token to the higher priority requester R before using it, as depicted in Figure 5b. Request R logs all intermediate nodes on the way and stops propagating when an edge along the probable owner path leads to an already logged node. Thus, request R stops propagating at T since T's probable owner is I (T sent the token to I) and the request had already passed I. In addition, a kill list of entries added to local queues on intermediate nodes is passed to the token requester R. Node R ensures that the request for mutual exclusion is only granted after this kill list has been received.

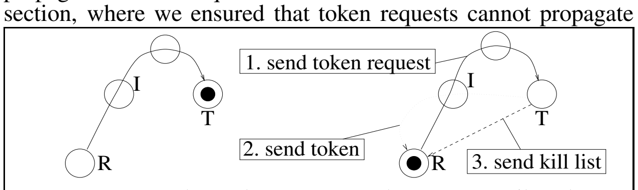
The added request on intermediate nodes will be processed when the token reaches the intermediate node the next time. There are two subcases. Either the token passes by intermediate node X before reaching R, for example node I in Figure 5b. Then, the request will be merged with other pending requests and it is locally recorded that the request has been served.

In the other case, the token passes by intermediate node X after reaching R. A kill list indicates the entries to be removed from local queues of all intermediates nodes. This list is received before or while R holds the token (indicated by the dashed edge in Figure 5b). When the token reaches a node in the kill list, entries of the local queue are removed if the kill list indicates this.

Submission of a kill list is also necessary as a result of a request by R that stops propagating when a higher or equally high priority request is pending at an intermediate node I (on the path to the token) whose probable owner is I. The request from R is queued locally on all intermediate nodes on the way to I including node I. In addition, a kill list is sent to R that includes all intermediate nodes up to I for this request. The token may now either be received by I, where it is immediately forwarded to R. Or another node on the way to I receives this token before I since I's request may have been issued very recently. In this case, the token is also forwarded to R by this other node, possibly even before reaching I. The kill list guarantees that all entries for R's request are eventually deleted from local queues of intermediate nodes, regardless of the location of the node between R and I.

5 Message Overhead

First, we analyze the number of messages necessary to register a single request for mutual exclusion. In an environment with n nodes, the algorithm requires an average of $O(log(n))$ messages for requests since the algorithm uses a logical tree similar to the algorithm by Naimi *at. el.* [16]. In fact, the prioritized algorithm reduces to an improved version of Naimi's algorithm when only one priority level is utilized. In the worst case, the links between nodes form a chain such that $n-1$ messages are required for the propagation of the request. This should be obvious from the last section, where we ensured that token requests cannot propagate



(a) Request Reaches Token (b) Request Trails Token

Figure 5. Abstract Model of Token Forwarding

in cycles, in other words, they cannot pass a node twice. There are at most two more messages required, one to send the token from an intermediate node to the requester and one to send the kill list (see Figure 5b). In the example, the token message from T to I is excluded from the message overhead of R's request since this message was caused by a different request, namely I's. More details about the overhead can be found in [14].

6 Performance Evaluation

On one hand, we want to investigate the performance of the prioritized algorithm. On the other hand, we want to compare the performance of the unprioritized algorithm by Naimi *et. al.* [16] with our prioritized algorithms when all requests are issued at the same priority. After all, the concept of local queues combined with local time-keeping may be used to enhance the unprioritized algorithm as well to reduce the amount of messages. Notice that logging of requests on intermediate nodes, as described in section 4, is not necessary in the absence of priorities.

6.1 Experimentation Environment

An algorithmic presentation of the prioritized protocol must be omitted due to lack of space but may be found in [14]. We implemented our algorithm and also an asynchronous version of the algorithm by Naimi *et. al.* [16] extended to reader and writer locks [9] that is described in more detail in [12]. The algorithms can both be used in our DSM-Threads environment [11] to guarantee mutual exclusion. This environment is similar to POSIX threads [20] in the sense that strict priority scheduling shall be enforced and FIFO scheduling within each priority level. However, distributed threads do not share any physical memory. Instead, a distributed virtual shared memory (DSM) is supported [9, 10]. The details of DSM-Threads are beyond the scope of this paper.

We designed a test environment along the lines of the work by Fu *et. al.* [5] measuring the *Number of Messages per Exchanged critical section entry* (NME) and the response time, *i.e.* the elapsed time between issuing a token request and the entry to the critical section. The measurements were taken for 8 tasks requesting 10 critical section entries each within varying intervals Γ between 5 and 900 milliseconds. The intervals were randomly distributed around Γ within ± 5 milliseconds. Within the critical section, a single write operation is performed to simulate access to a shared variable. If a node releases the lock of a critical section and, due to lack of contention, reacquires it before token forwarding, the measurement is ignored since remote overhead should be measured.

The DSM-Threads environment currently runs on SPARC and Intel x86 platforms. Thus, we chose a set of SPARC workstations connected via regular Ethernet to gather performance numbers. This low-speed network allowed us to gather measurements for a relative comparison between the two algorithms. We designed three different experiments.

The first one uses 8 threads within one process on a SPARC 20 at 60 MHz, where message passing is simulated by buffers within physically shared memory, and simulates a high-speed network with extremely low latencies exposing the overhead in bookkeeping of our approach. The second one measures the performance for 8 processes on a dual processor SPARC 20 at 60 MHz using TCP messages simulating a medium-speed network. And the third one distributes 8 processes onto different workstations running at processor speeds between 40 and 80 MHz, again with TCP messages, representing an actual low-speed network. TCP messages may be processed in a different order in that they were received due to the multi-threading of our environment. This allows us to test the protocol for out-of-order message delivery. The overhead of sending a single TCP message, including the overhead of the DSM-Threads message layer, ranges between 10 and 13 milliseconds. We have not yet tuned our message passing layer.

6.2 Measurements

Figure 6 covers the first experiment with simulated communication via physically shared memory. It illustrates the overhead of our protocol over Naimi's approach. About three messages are required for mutual exclusion, except for short critical section intervals Γ where two messages suffice. Short intervals between requests result in chains of owners for both algorithms since requests arrive faster than they can be serviced. Hence the high response time for small Γ. The response time of Naimi's algorithm is about one third of the response time of our algorithm, on the average. Since we simulate message passing, the difference in response time represents the additional computation overhead required by our protocol. This additional cost is caused by larger messages that have to be processed after transmitting them. The savings due to local queues are not visible since the latency of messages in this simulation environment is extremely short. Of course, our approach adds functionality.

Figure 7 covers the second experiment of TCP communication between processes on one machine. The number of messages is smaller for our algorithm, in particular for short interval request times. This behavior can be attributed to our local queues. The response time is still smaller for Naimi's algorithm but its overhead has risen to about three fourth of our algorithm, due to the difference in the number of messages. When requests are issued frequently (small Γ), the response time is relatively high. As contention recedes for larger Γ, the response time becomes smaller.

Figure 8 depicts the third experiment of TCP communication between processes on different workstations. The number of messages is almost the same for both protocols. The response time of Naimi's approach is about half that of our approach. For short critical section intervals, we observed that many locks were acquired locally before a remote request required token forwarding. Thus, the results for small Γ may be misleading due to small sample sizes. Once the algorithms require three messages per mutual exclusion, Naimi's algorithm has a response time of about two thirds of our approach.

In summary, our approach has additional functionality but this comes at a price of additional processing cost and longer response times. If one does not need priority support, Naimi's algorithm enhanced by local queues may be a good choice. In a multi-threaded environment, our algorithm may increase the exploitation of potential parallelism. For strict priority scheduling of requests for mutual exclusion, our algorithm should be a good choice.

7 Related Work

A number of algorithms exist to solve the problem of mutual exclusion in a distributed environment. Chang [2] and Johnson [7] give an overview and compare the performance of such algorithms. Goscinski [6] proposed a priority-based algorithm based on broadcast requests using a token-passing approach. Chang [1] developed extensions to various algorithms for priority handling that use broadcast messages [19, 18] or fixed logical structures with token passing [17]. Chang, Singhal and Liu [3] use a dynamic tree similar to Naimi *et. al.* [15, 16]. In fact, the only difference between the algorithms seems to be that the root of the tree is piggybacked in the former approach while the latter (and older) one does not use piggybacking. Due to the similarity, we simply referred to the older algorithm in this paper. Other mutual exclusion algorithms (without token passing) employ global logical clocks and timestamps [8]. These algorithms can be readily extended to transmit priorities together with timestamps. However, all of the above algorithms, except Raymond's and Naimi's, have a message complexity larger than $O(log(n))$ for a request. Finally, Raymond's algorithm uses a fixed logical structure while we use a dynamic one to allow the addition and deletion of nodes. Furthermore, Raymond needs $O(log(n))$ messages to send the token to a requester, where our algorithm only requires either one or two messages (at the same priority level). The modified version of Raymond's algorithm by Fu *et. al.* [5] is, in its essence, similar to our local queues but with just one entry. Finally, all of the previous algorithms use synchronous message passing and assume a single request per node at a time while our algorithm works asynchronously with multiple requests per node to provide more concurrency within a multi-threaded environment.

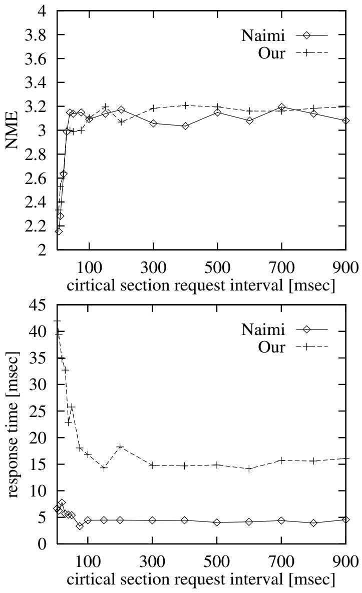


Figure 6. Simulated Comm., via Physically Shared Memory

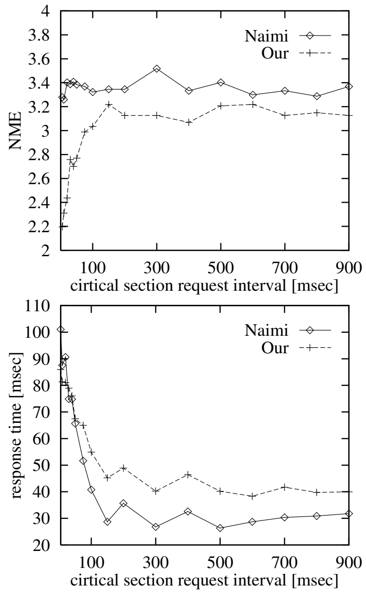


Figure 7. TCP Comm., SMP with 2 Processors

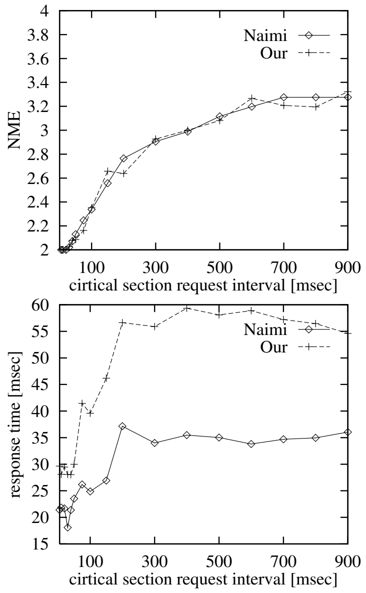


Figure 8. TCP Comm., Distributed (Different Workstations)

8 Conclusion

The main contribution of this paper is a prioritized algorithm for distributed mutual exclusion with an average message overhead of $O(log(n))$ and a worst-case overhead of $n + 1$ messages per request, respectively. The algorithm is based on a token-passing scheme and uses local queues, which together form a global queue of pending requests. The queues are ordered by priority and then FIFO within each priority level. We give a framework to relate requests of different local queues to each other in terms of FIFO ordering by using only local time-keeping instead of a global logical clock with timestamps. Thus, local queues can be merged while preserving FIFO ordering. We utilize a set of logical trees with associated priority levels to let requests propagate either up to the token holder or to another requester with higher priority. The resulting algorithm for prioritized mutual exclusion requires forwarding of queues and local bookkeeping. It works for out-of-order message delivery and supports asynchronous message handling of multiple requests from one node if nodes are multi-threaded. The algorithm has the same message complexity as its best known non-prioritized counterparts while previous prioritized algorithms have a higher message complexity, to our knowledge. Furthermore, the concept of local queues combined with local time-keeping can be incorporated into arbitrary token-based protocols with or without priority support to reduce the amount of messages. Performance results indicate that the additional functionality of our algorithm comes at the cost of 30% longer response times within our test environment for distributed execution when compared with an unprioritized algorithm. This result suggests that the algorithm should be used when strict priority ordering is required.

References

[1] Y. Chang. Design of mutual exclusion algorithms for real-time distributed systems. *Journal of Information Science and Engeneering*, 10:527–548, 1994.

[2] Y. Chang. A simulation study on distributed mutual exclusion. *J. of Parallel and Distributed Computing*, 33(2):107–121, Mar. 1996.

[3] Y. Chang, M. Singhal, and M. Liu. An improved O(log(n)) mutual exclusion algorithm for distributed processing. In *Parallel Processing*, volume 3, pages 295–302, 1990.

[4] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, Aug. 1991.

[5] S. Fu, N. Tzeng, and Z. Li. Empirical evaluation of distributed mutual exclusion algorithms. In *International Parallel Processing Symposium*, pages 255–259, 1997.

[6] A. Goscinski. Two algorithms for mutual exclusion in real-time distributed computer systems. *Journal of Parallel and Distributed Computing*, pages 77–82, 1990.

[7] T. Johnson. A performance comparison of fast distributed mutual exclusion algorithms. In *International Conference on Parallel Processing*, pages 258–264, 1995.

[8] L. Lamport. Time, clocks and ordering of events in distributed systems. *Communication of the ACM*, 21(7):558–565, June 1978.

[9] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. of Computer Syst.*, 7(4):321–359, Nov. 1989.

[10] V. Lo. Operating systems enhancements for distributed shared memory. *Advances in Computers*, 39:191–237, 1994.

[11] F. Mueller. Distributed shared-memory threads: DSM-threads. In *Workshop on Run-Time Systems for Parallel Programming*, pages 31–40, Apr. 1997.

[12] F. Mueller. On the design and implementation of DSM-threads. In *Int. Conference on Parallel and Distributed Processing Techniques and Applications*, pages 315–324, June 1997. (invited).

[13] F. Mueller. Prioritized token-based mutual exclusion for distributed systems. In *Workshop on Parallel and Distributed Real-Time Systems*, Apr. 1997.

[14] F. Mueller. Prioritized token-based mutual exclusion for distributed systems. TR 97, Inst. f. Informatik, Humbolt University Berlin, Jan. 1998. www.informatik.hu-berlin.de/˜mueller.

[15] M. Naimi and M. Trehel. An improvement of the log(n) distributed algorithm for mutual exclusion. In *Distributed Computing Systems*, 1987.

[16] M. Naimi, M. Trehel, and A. Arnold. A log(N) distributed mutual exclusion algorithm based on path reversal. *JPDC: Journal of Parallel and Distributed Computing*, 34(1):1–13, Apr. 1996.

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

[18] M. Singhal. A heuristically-aided algorithm for mutual exclusion in distributed systems. *IEEE Transactions on Computers*, 38(5):651–662, May 1989.

[19] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *ACM Transactions of Computer Systems*, 18(12):94–101, Dec. 1993.

[20] Technical Committee on Operating Systems and Application Environments of the IEEE. *Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)*, 1996. ANSI/IEEE Std 1003.1, 1995 Edition, including 1003.1c: Amendment 2: Threads Extension [C Language].