Implementing Distributed Mutual Exclusion on Multithreaded Environments: The Alien-Threads Approach

Federico Meza[1], Jorge Pérez R.[1], and Yadran Eterovic[2]

[1] Depto. de Ingeniería de Sistemas, Universidad de Talca
Camino Los Niches Km. 1, Curicó – CHILE
[fmeza,jperez]@utalca.cl
[2] Depto. de Ciencia de la Computación, Pontificia Universidad Católica de Chile
Casilla 306 – Santiago 22, Santiago – CHILE
yadran@ing.puc.cl

Abstract. We present a simple implementation of a token-based distributed mutual exclusion algorithm for multithreaded systems. Several per-node requests could be issued by threads running at each node. Our algorithm relies on special-purpose *alien threads* running at host processors on behalf of threads running at other processors. The algorithm uses a tree to route requests for the token. We present a performance simulation study comparing two versions of our algorithm with a known algorithm based on path reversal on trees. Results show that our algorithm performs very well under a high load of requests while obtaining acceptable performance under a light load.

Keywords. Distributed mutual exclusion, multithreading, parallel programming, concurrent programming, distributed shared memory.

1 Introduction

Mutual exclusion aims to provide synchronized access to shared resources ensuring that, at any time, at most one process can be executing in its critical section. Distributed mutual exclusion algorithms focus on mutual exclusion on distributed environments lacking shared memory. Several algorithms address the distributed mutual exclusion problem for systems where only one process is running at each processor. Multithreaded distributed systems allow the existence of several threads of execution within each distributed process. Thus, there is a need to provide mutual exclusion to a large number of distributed threads. We are particularly interested in Distributed Shared Memory systems [1] with support for multithreading and thread migration.

In this work, we present an algorithm for distributed mutual exclusion in a multithreaded system. The algorithm is token-based, and it uses a tree to route requests issued to acquire the token. We rely on special-purpose *alien threads* running at host processors on behalf of threads running at other processors.

Federico Meza was supported by Fondecyt under grant 2990074.

When a thread asks for permission to enter its critical section, if the token is not present at the processor it is running at, a remote alien thread is activated in order to obtain the token and send it to the requesting processor. Alien threads behave just like ordinary threads, and must compete for the token with other user threads. Thus, our algorithm is simple but correct.

We performed a simulation study comparing two versions of our algorithm with a previously proposed algorithm based on path reversal on trees. This algorithm is, to the best of our knowledge, the only documented implementation addressing the same problem. Results obtained from the simulation are encouraging. Our algorithm performs very well under high load conditions, outperforming the other proposal as the number of threads per node increases.

Our algorithm was successfully implemented on *DSM-PEPE*, a multithreaded distributed system with support for thread migration [2].

2 The algorithm

2.1 System model

The system is a loosely-coupled network of computers, consisting of n processors: $p_1, p_2, ..., p_n$. At any time, at each processor p_i, there are m_i threads running. Threads are allowed to migrate according to some system policy, for instance, pursuing load balancing or minimal message exchange.

Processors communicate through message passing. We assume that message delivery is guaranteed by the network. We also assume that two messages issued at one processor and addressed to the same node are received in the same order at the destination. This is the usual behavior of switched local area networks, where there is only one possible route between each pair of computers.

A thread wishing to enter its critical section must obtain permission by calling `Acquire()`. The thread could be delayed until mutual exclusion can be guaranteed. Once the thread leaves the critical section, it must notify the system by calling `Release()`. Mutual exclusion must be ensured between the call to `Acquire()` and the call to `Release()`.

2.2 Brief description

Our algorithm is token-based. A thread wishing to enter its critical section must obtain a single system-wide token. Uniqueness of the token guarantees the mutual exclusion [3]. At the higher level, ownership of the token is not granted directly to threads but to processors. Once a processor owns the token, threads running at that processor can compete for it. Requests issued at each processor are stored in a local queue owned by the processor. Ownership of the token is granted to one of the processors during initialization. For the remaining processors a unique path must exist to allow them to reach the actual owner of the token. This is accomplished through a chain of *probable owners*, building up a tree rooted at the first owner.

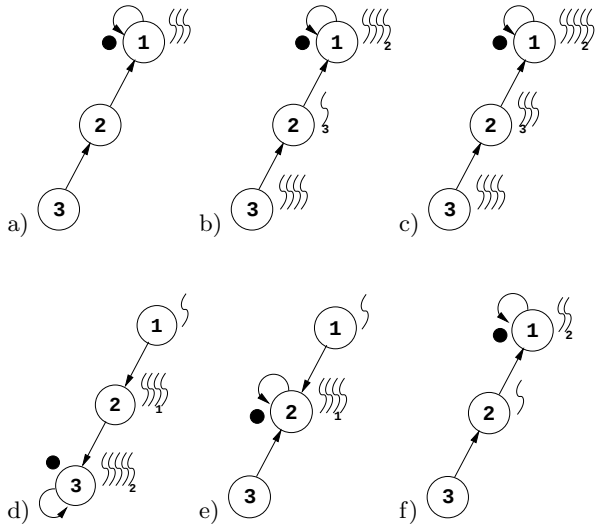

Fig. 1. Behavior of the algorithm when servicing several requests

Requests made by threads running at the processor currently owning the token are serialized and served according to their arrival time. A request issued by a thread running at a processor not owning the token involves sending a request to the probable owner. Our algorithm accomplished this task, by signaling a special-purpose thread running at the probable owner of the token.

At any processor, there are $n-1$ *alien threads*, each acting on behalf of one of the remaining $n-1$ processors in the system. Alien threads behave just like ordinary threads, but they are blocked most of the time. An alien thread at processor p_i is signaled when some thread at the home processor –that is, the processor for which the alien thread is working on behalf of– is requiring the token, and it is presumed that the token is held by processor p_i, that is, the probable owner at the home processor is set to p_i. The woken alien thread asks for the token at its host processor. Then, once the token is granted, the alien thread sends the token to the processor it is representing.

Note that it is possible for a woken alien thread to find out that the token is no longer on its host processor. When this situation occurs, the alien thread acts like an ordinary thread requesting the token. The alien thread signals the alien thread on the processor where the token apparently went to, that is, the probable owner on its host processor. This scenario could appear several times, until the processor currently holding the token is reached. This kind of forwarding resembles the algorithm described by Raymond for distributed mutual exclusion of single-threaded processors [4].

Figure 1 shows the behavior of our algorithm in a system with 3 nodes. Initially, node 1 owns the token –represented as a filled circle– and several threads are blocked at that node, each waiting to enter its critical section (Figure 1a).

Note that node 1, the owner of the token, is at the root of the tree used to route the requests. At this moment, several requests are issued from threads running at node 3. The first of these requests produces a signal to an alien thread on node 2, to wake up and act on behalf of node 3 (Figure 1b). However, since the token is not present at 2, the recently woken alien thread blocks, producing a signal to an alien thread on node 1, to wake up and act on behalf of node 2. This alien thread waits at the end of the local queue of node 1. Note that the request made by a thread running at node 3 produced two requests at remote nodes, issued by alien threads. After that, some threads issued new requests at nodes 1 and 2, producing their local enqueueing (Figure 1c). At this point it is important to note the behavior of the alien threads currently active in the system. An alien thread is waiting for the token at node 1, the current owner of the token, on behalf of 2, and another alien thread is waiting for the token at node 2, on behalf of 3. The thread that issued the original request is waiting for the token at node 3. A node only sees a queue of requests, some issued by local threads and some issued by alien threads. Eventually, the alien thread running at node 1 on behalf of 2 obtains the token, and sends it to node 2. Since the queue was not empty at this time, that is, there are pending requests at node 1, an alien thread on node 2 is signaled to bring the token back to node 1. At the head of the queue at node 2 was the alien thread that acts on behalf of node 3, so it sends the token to node 3, and an alien thread on node 3 is signaled to bring the token back to node 2. Note that, at node 2 there are several pending requests, including one that will bring the token back to node 1 (Figure 1d). Eventually, the token returns to node 2 (Figure 1e) and to node 1 (Figure 1f).

2.3 Detailed description

Each processor must hold the following information:

- `probOwner`: process identifier –pid– of the processor last known as the token owner. Initially set in such a way that there is a single path, following the probOwners chain, from each node to the initial owner of the token. The first owner sets probOwner to its pid.
- `tokenRequested`: `true` if there are pending requests for the token issued from this processor, *i.e.*, a request for the token has been already sent. Initially `false` at every processor.
- `numLocal`: number of requests for the token that have been issued locally; initially 0 at every processor. Recall that only the first request actually makes an alien thread to be signaled.

Local mutual exclusion for the operations showed below is mandatory. However it has been omitted intentionally to illustrate the solution more clearly. Semantics of the *wait* and *signal* operations are consistent with those on conditions variables. A signal across processors involve sending a message to the target processor.

A thread acquiring the token must execute:

```
Acquire() {
   numLocal++;
   if (probOwner != pid) && (! tokenRequested) {
      // Not owner and not previously requested => Request token
      signal(alien thread on probOwner);
      tokenRequested = true;   // to avoid multiple requests
      wait(for signal from the alien thread);
      probOwner = pid;        // processor becomes owner
      tokenRequested = false;
   }
   else {
      // Processor owns token, or token has been requested already
      if (numLocal > 1) {
         wait(for signal from another local thread);
      }
   }
}
```

If the token is not currently held by the processor it must be requested, by signaling the alien thread on the probable owner. The thread blocks waiting for the token to arrive. If some other thread has previously called `Acquire`, we must prevent multiple requests. If the token is held by the processor, or it has been requested already, there is no need for remote requests. If the token is held but free the thread is allowed to enter its critical section.

A thread releasing the token must execute:

```
Release() {
  numLocal--;
  signal(local thread waiting for the token);
}
```

Note that the thread being signaled could be a user thread or an alien thread.

The alien thread executing on processor *host* on behalf of processor *home* must execute the following code:

```
alienThread(host_pid, home_pid) {
  while(true) {
    wait(for signal from home processor);        // stay idle
    Acquire();             // acquire token on host processor
    signal(thread waiting for the token on the home processor);
    probOwner = home_pid;  // update host-processor's probOwner
    numLocal--;
    if (numLocal > 0) {
      // Request the token on behalf of host processor
      tokenRequested = true;   // to avoid multiple requests
      signal(alien thread on home processor on behalf of host);
    }
  }
}
```

An alien thread waits until signaled from its home processor. Then, it acquires the token, competing with local threads on the host processor, as well as with other alien threads trying to get the token on behalf of their homes. Once an alien thread succeeded on acquiring the token, it signals its home processor, allowing a remote waiting thread to resume under mutual exclusion. It is possible to have additional threads left on the local queue when an alien thread acquires the token delivering it to its home processor. If this happens, the alien thread requests the token before turning idle. This is accomplished by signaling the alien thread that represents its host on its home processor. A simple improvement to the algorithm presented involves *piggybacking* this request on the same signaling message that delivers the token.

The behavior of an alien thread holding the token is slightly different from the behavior of a user thread. A user thread is expected to release the token once it leaves the critical section. However, an alien thread does not release the token, but delivers it directly to another thread at his home processor instead.

2.4 A variant on the proposed algorithm

An alien thread forwards requests when the token is not present at its host processor. This is done by signaling the alien thread on the probable owner, on behalf of its host processor. Thus, the token is forced to follow exactly the same path followed by the requests. This behavior is desirable under high requests load, because there will be always pending requests on the returning path of the token, avoiding the exchange of extra messages. However, under a light load, the token could be sent directly to the processor that issued the first request, avoiding the extra steps produced by the forwarding. Only the code executed by the alien threads must be modified in order to implement this variant:

```
alienThread(host_pid, home_pid) {
  while(true) {
    wait(for signal from home processor);         // stay idle
    if ((probOwner != host_pid) && (numLocal == 0)) {
      signal(alien thread on behalf of home, on host's probOwner);
    }
    else {
      // Behaves like the original alien thread
      Acquire();             // acquire token on host processor
      signal(thread waiting for the token on the home processor);
      probOwner = home_pid;  // update host-processor's probOwner
      numLocal--;
      if (numLocal > 0) {
        // Request the token on behalf of host processor
        tokenRequested = true;   // to avoid multiple requests
        signal(alien thread on home processor on behalf of host);
      }
    }
  }
}
```

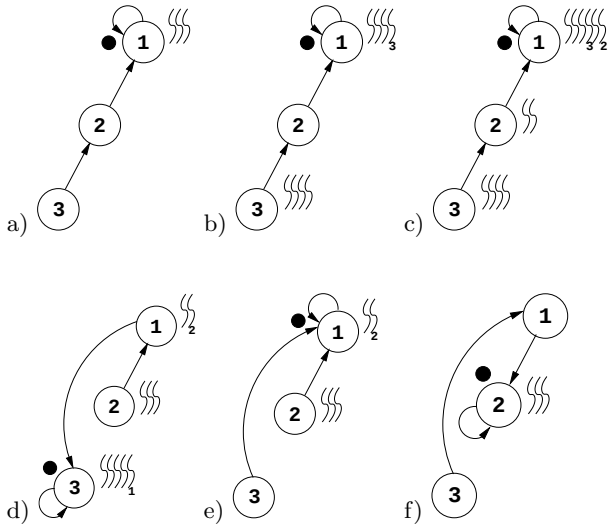

Fig. 2. Behavior of the modified algorithm when servicing several requests

Instead of simply forwarding requests, the improved alien thread checks first if there are local requests at its host processor that justify acquiring the token. Otherwise, it signals an alien thread on behalf of its home processor, avoiding the passing of the token across its host processor.

Figure 2 shows the behavior of the modified algorithm for the same requests sequence used in the example of the original algorithm in Section 2.2. Note that, when the first request by a thread running at node 3 is issued, the alien thread at node 2 does not remain active because the queue at node 2 is empty. It just forwards the signal to an alien thread at node 1, to wake up and act on behalf of node 3 instead (Figure 2b). Eventually, this alien thread obtains the token, and sends it directly to node 3 (Figure 2d).

The path followed by the token back to the requester node is not necessarily the same path previously followed by the request on its way to the owner of the token. Note that several alien threads in the request path just forwarded the request, changing the topology of the tree.

3 Proof outline

A mutual exclusion algorithm must satisfy several conditions. The following is an outline of the proof of correctness for three of these conditions, considering the original algorithm.

Mutual exclusion: It must be assured that, at any time, at most one thread can be executing in its critical section. Our algorithm is token-based: there is a single system-wide token, owned by the node having `probOwner == pid`. This

condition is enforced during initialization. A thread asks for the token by executing `Acquire` and could be delayed on two conditions: (1) when the token is currently held by its host node but it is assigned to another thread, or (2) when the token is not locally present at the time. Either way, the thread is allowed to continue executing in its critical section only when the thread currently holding the token relinquished it by executing `Release`, or when an alien thread signals the blocked thread remotely. In the former, it is straightforward to verify that mutual exclusion is assured. In the latter, the signaling alien thread previously obtained the mutual exclusion by executing `Acquire` on its host processor. Since an alien thread does not have a critical section, it relinquishes the mutual exclusion on behalf of the thread that made the request on its home processor. This way, mutual exclusion among threads is assured.

Deadlock freedom: It is easy to verify that a deadlock can not occur under some reasonable restrictions. For a deadlock to occur there must be a circular-wait condition involving two or more threads in the system. Assuming that a thread executing in its critical section is not allowed to execute `Acquire` again, this condition will never occur.

Starvation freedom: If we assume the use of a fair policy for serving local requests at each node starvation will not occur. Recall that we have a single path from each node to the node currently holding the token. Note that a request issued at a node not owning the token results in an alien thread being queued at the node currently owning the token. If the local service policy is fair, the alien thread eventually obtains the token and allows the thread it is acting on behalf of, to enter its critical section.

4 Performance

Analytic studies of distributed mutual exclusion algorithms are hard to perform, due to the rapid growth of the cardinality of the state space as the number of nodes increases [5]. In multithreaded systems, the size of the state space grows even faster. For this reason, we choose a simulation approach to study the performance of our proposal.

4.1 Simulation model

We use a simulation model based on similar studies [5], [6]. We assume that requests to enter the critical section arrive at each node according to a Poisson process with parameter λ. Thus, the time elapsed between critical section requests behaves according to an exponential distribution. We assume that, at every node, requests are made by randomly-chosen user threads. It is important to note that the simulation process remains under Poisson behavior as long as any running –not waiting– local thread exists in a node. When all the threads running at a node are waiting to enter to its critical section, the process stops until the first local thread completes its critical section.

The λ parameter will give us a notion of the load of the entire system. The time taken by a thread to execute its critical section is modeled as a constant C. The message propagation delay is a constant M multiplied by a random number having a uniform distribution between 0 and 1.

We are interested in two main measures: the average number of messages exchanged per critical section entry, and the average waiting time for the permission to enter the critical section.

To obtain statistically reliable results we made long-time simulations executing $100,000$ critical section entries. On each experiment we use $N = 31$ nodes, because we have a binary complete tree for the initial state. We simulate a variable number of threads per node. The parameter λ takes values in the $[0, 1]$ interval. The parameter M was taken as 0.1 and the parameter C as 0.01. These values are consistent with those used in similar studies [5],[6].

4.2 Results

Figures 3 through 5 show the results of the simulations for the two versions of our algorithm –using piggybacking– as well as the algorithm proposed by Mueller [7], using 1, 5 and 10 threads per node. In Figure 3 there is a single thread per node. In this case, the first implementation of our algorithm resembles the algorithm by Raymond [4]. Obtained results are consistent with that fact [6].

Under a light load, the second version of our algorithm requires fewer messages than the first one, because the token is sent from the releaser node to the requester directly. The alien thread that was waiting for the token at the releaser node, acts on behalf of the requester node. The first implementation enforces the token to travel along the tree structure to reach the requester node. This is so because several alien threads need to be signaled in the path previously followed by the requests. The algorithm proposed by Mueller has the best comparative performance under a light load, considering the number of messages exchanged. This is due to the *aggressive path compression* technique, characteristic of the path reversal approach [8].

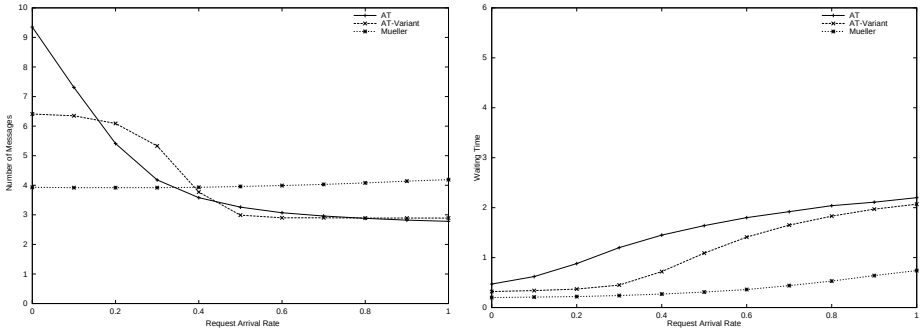

Fig. 3. Performance comparison of the 3 algorithms with a single thread per node.

Considering the waiting time, the second version of our algorithm behaves better than the first one under all loads. The algorithm proposed by Mueller outperforms the other two.

Under a high load, both versions of our algorithm need almost the same number of messages per critical section. When the first alien thread associated to a request is signaled, it is most likely that the token was already requested on the host node. This will make both versions of the algorithm behave the same way. This situation can be easily observed in the code of the alien thread: both versions will execute the same code under high load conditions. This is also the cause of the very small number of messages exchanged per critical section under high loads. Both implementations of the alien thread algorithm outperform Mueller's algorithm.

When we turn to multithreaded scenarios (Figures 4 and 5) the relative behavior among the two versions of our algorithm remains unchanged. Moreover, the waiting time is almost identical, for every number of threads per node. Besides, an important decrease in the number of messages exchanged per critical section entry is achieved under high loads as the number of per-node threads increases. Once a request has been sent from a node –that is, an alien thread has been signaled–, subsequent requests issued by threads on the home node do not involve the sending of additional messages. Thus, most of the requests issued by threads wishing to enter the critical section will be served without message exchange. The algorithm proposed by Mueller does not show any of these behaviors. The number of messages exchanged does not change significantly as the number of threads per node increases. Waiting time increases abruptly as the system load increases. The growing rate of the waiting time in the Mueller's algorithm, also increases as the number of threads per node increases. His algorithm is very sensitive to load growth on multithreaded scenarios.

The second version of the alien thread algorithm outperforms the initial version in all aspects. The algorithm proposed by Mueller showed better performance under a light load. Under high loads, the alien thread algorithm showed better results.

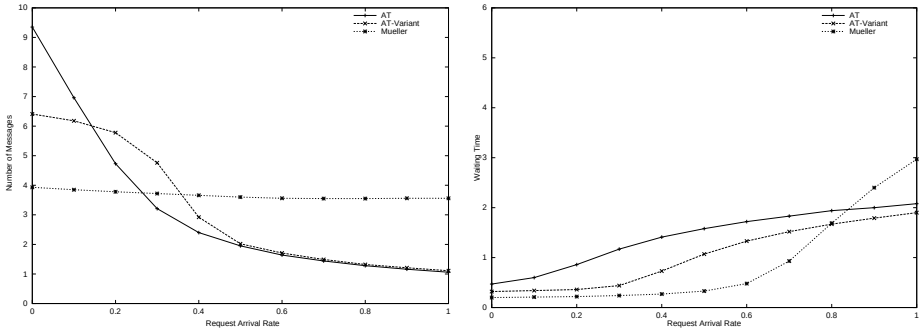

Fig. 4. Performance comparison of the 3 algorithms with 5 threads per node.

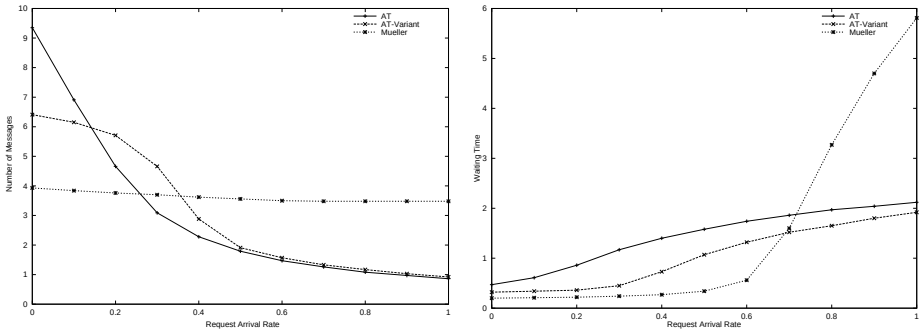

Fig. 5. Performance comparison of the 3 algorithms with 10 threads per node.

5 Related work

Several distributed mutual exclusion algorithms have been proposed in the literature. They can be classified as *permission-based* or *token-based* [9]. We focus our study on token-based distributed algorithms, excluding those algorithms that use a central coordinator.

Token-based algorithms rely on a unique token which must be acquired by a process wishing to enter its critical section. The token could be traveling from one process to another continuously or could be obtained by sending a request. The algorithms proposed by Raymond [4], by Neilsen and Mizuno [10], by Banerjee and Chrysanthis [11], and by Naimi, *et al.* [8] fall into this category.

Distributed mutual exclusion for multithreaded environments has not been studied extensively. The design and implementation of distributed synchronization primitives are presented by Mueller, focusing on the impact of multithreading on synchronization [7]. Distributed mutual exclusion is based on a token-passing mechanism based on the algorithm described by Naimi, *et al.* [8].

6 Conclusions

We presented a simple implementation of a token-based algorithm providing mutual exclusion to distributed threads running on a loosely-coupled system. This mechanism has been successfully implemented on a Distributed Shared Memory system supporting thread migration.

We developed two versions of the algorithm and compare them to a known implementation of another algorithm, targeting to the same problem. A simulation of performance shows that both of our algorithms outperforms the other implementation under high load conditions. The difference increases as the number of threads per node increases. Under a light load, our algorithms still perform within reasonable limits.

The first version of our algorithm, limited to a single user thread per node, behaves just like a well-known single-threaded distributed mutual exclusion algorithm [4]. The third algorithm considered in our study [7], was originally conceived as an extension of another single-threaded algorithm based on path reversal on trees [8]. Our intention is to extend the study to several single-threaded algorithms for distributed mutual exclusion, exploring the feasibility of extend them using the same ideas used to develop the alien-threads algorithm.

References

1. Li, K., Hudak, P.: Memory Coherence in Shared Virtual Memory Systems. ACM Transactions on Computer Systems **7** (1989) 321–359
2. Meza, F., Campos, A.E., Ruz, C.: On the Design and Implementation of a Portable DSM System for Low-Cost Multicomputers. In: Proc. of the Intl. Conference on Computational Science and its Applications (ICCSA 2003). Volume 2667 of Lecture Notes in Computer Science., Springer (2003)
3. Hélary, J.M., Mostefaoui, A., Raynal, M.: A General Scheme for Token- and Tree-Based Distributed Mutual Exclusion Algorithms. IEEE Transactions on Parallel and Distributed Systems **5** (1994) 1185–1196
4. Raymond, K.: A Tree-Based Algorithm for Distributed Mutual Exclusion. ACM Transactions on Computer Systems **7** (1989) 61–77
5. Chang, Y.: A Simulation Study on Distributed Mutual Exclusion. Journal of Parallel and Distributed Computing **33** (1996) 107–121
6. Johnson, T.: A Performance Comparison of Fast Distributed Mutual Exclusion Algorithms. In: Proceedings of the 9th International Symposium on Parallel Processing (IPPS'95), Los Alamitos, CA, USA, IEEE Computer Society Press (1995) 258–264
7. Mueller, F.: Decentralized Sinchronization for Multithreaded DSM. In: Proc. of the 2nd. Workshop on Software Distributed Shared Memory (WSDSM 2000). (2000)
8. Naimi, M., Trehel, M., Arnold, A.: A $log(N)$ Distributed Mutual Exclusion Algorithm based on Path Reversal. Journal of Parallel and Distributed Computing **34** (1996) 1–13
9. Raynal, M.: A Simple Taxonomy for Distributed Mutual Exclusion Algorithms. ACM SIGOPS Operating Systems Review **25** (1991) 47–50
10. Neilsen, M., Mizuno, M.: A DAG-Based Algorithm for Distributed Mutual Exclusion. In: Proc. of the 11th. International Conference on Distributed Computing Systems (ICDCS 96). (1991) 354–360
11. Banerjee, S., Chrysanthis, P.: A New Token Passing Distributed Mutual Exclusion Algorithm. In: Proc. of the 16th. International Conference on Distributed Computing Systems (ICDCS 96). (1996) 717–725