Efficient distributed Synchronization within an all-software DSM system for clustered PCs[1]

Sven M. Paas, Karsten Scholtyssik
Lehrstuhl für Betriebssysteme, RWTH Aachen
Kopernikusstr. 16, D-52056 Aachen, Germany
Tel.: +49-241-80-7634, Fax: +49-241-8888-339
e-mail: `contact@lfbs.rwth-aachen.de`

Abstract. *We propose a new extension to Chang, Singhal and Liu's scalable algorithm for distributed mutual exclusion in order to allow more than one process at a given time to enter a distributed critical section. This is done by integrating distributed single writer / multiple reader semantics into the original algorithm without affecting the performance for exclusive lock requests. We show that software subsystems providing the shared virtual memory abstraction as well as parallel applications built on it benefit from the new extension significantly. First implementation results employing a newly developed all-software, page based distributed shared memory (DSM) system for clustered Windows NT systems are presented.*

Keywords: *distributed synchronization, shared virtual memory, cluster computing*

1 Introduction

Distributed mutual exclusion algorithms [DI89, MA92, SR92, SI93] solve the problem of mutual exclusion for distributed environments. Whenever a process has to read or update a certain shared resource like data structures, files, shared virtual memory pages, it first enters a *distributed critical section* to achieve *distributed mutual exclusion* to ensure that no other process will access the shared resource at the same time.
On uni-processor or symmetrical multiprocessing (SMP) systems, the operating system usually provides a whole family of process or thread synchronization primitives [SI94]. Besides pure mutual exclusion functionality, there are primitives providing „weaker" synchronization semantics in order to let more than one process at a given time enter its critical section. A special case is given by so called *single writer / multiple reader* locks [KR93]: they allow a single writer, but multiple concurrent readers in the critical section at a given time. Depending on the access pattern on the shared resource, this synchronization primitive can drastically improve the performance of the application by allowing a higher potential degree of parallelism [CH94].
In distributed memory systems, on the other hand, message passing based algorithms are used to achieve distributed mutual exclusion. This idea was introduced by Lamport [LA78] and constantly improved over the last decade [MA85, MJ85, SU85, NA87, RA89, CH90]. While there exist fast and efficient algorithms for distributed synchronization [CH96, RA94, JO94], these approaches are unnecessary restrictive with respect to lock utilization when a large fraction of all requests are read-only.

[1] This work was partly funded by Intel Corporation.

1.1 Assumptions

We define the following system requirements for the algorithms presented in the following sections:

- We have a distributed system with no shared memory (neither physical nor virtual).
- Communication is solely done by passing messages between processors (also called nodes).
- The network of processors is fully connected and reliable. Thus, each processor can communicate directly and securely with each other processor.

It is important to understand that message exchanging itself always represents a synchronization point [DI89] - implementing a distributed synchronization algorithm on top of message passing basically means to provide an efficient and correct *message passing protocol* that guarantees distributed mutual exclusion or some other sort of synchronization semantics. Thus, the main duty of a distributed synchronization operating system service is to preserve the application programmer from using explicit message passing for distributed synchronization.

1.2 Overview of this paper

In the next section, we will informally specify reader / writer lock semantics for distributed environments and recall related work in this area. Motivated by a fast available distributed mutual exclusion algorithm presented originally by Chang, Singhal and Liu [CH90], we propose an extension to their efficient path compression technique to support concurrently executing distributed readers in section 3. We evaluate the extension of the algorithm by a newly developed page based DSM system for Windows NT systems. The paper closes with a summary in section 5.

2 Distributed Reader / Writer Lock Semantics

Basically, a distributed reader / writer synchronization primitive (also called *lock*) can be derived from the usual uni-processor (or SMP) case, where the underlying operating system handles all synchronization requests in a centralized manner. If we use a token based distributed mutual exclusion algorithm [SI93], we define the following functionality for a protocol extension providing single writer / multiple reader semantics on a distributed system:

- *Write request functionality* is inherited by the usual protocol. Acquiring a token for writing basically means acquiring the token exclusively. A token can be granted for writing if and only if no other process possesses the token (whether for writing or for reading).
- *Read request functionality* is added to the usual protocol. Acquiring a token for reading means creating a virtual copy of the token and granting it to the requesting process. A copy of the token can be granted for reading, if no other process possesses the token exclusively for writing. Multiple processes may acquire the copies of the token for reading concurrently.

While there exist efficient algorithms providing the above synchronization semantics for shared memory environments [KR93], the only known approach for distributed environments

was presented by Johnson [JO96]. In contrast to [JO96], we do *not* require a restriction in the order of the distributed processes‘ requests to leave a critical section. That is, any process should be able to acquire or to immediately release a distributed reader / writer lock at any given time like in the uni-processor case.
In order to prevent processes which want to acquire the token for writing from starving, it is very important that the requests to the token are scheduled in a “fair” manner. Starving may occur by other processes constantly requesting and releasing the lock for reading. The most often implemented fair scheduling strategy for lock request is strict FCFS (*first come first served*) order. In principle, it is then possible to implement distributed single writer / multiple reader lock semantics on top of a distributed mutual exclusion algorithm. There are at least two straightforward possibilities to do that:

- First, each reader could be associated with a dedicated distributed exclusive lock. That is, each processor i (where $i = 1 ... N$, and N is the number of processors) which acquires the lock for reading has to acquire an associated lock L_i before it can enter its distributed critical section. Each process j which acquires the lock for writing has to acquire *all* locks L_k ($k = 1 ... N$). The disadvantage of this scheme is that acquiring a token for writing is a very expensive operation.
- Second, a virtually shared integer *cnt* variable containing the current number of readers protected by one distributed exclusive lock could be implemented. Each reader increments this variable, while each writer has to wait for *cnt* to become zero. The disadvantage of the scheme is that the shared *cnt* variable is constantly polled by the requesters, causing unnecessary network overhead.

Instead of trying to implement distributed single writer / multiple reader semantics on top of an existing algorithm for distributed synchronization, we propose to modify and extend these protocols to support concurrent readers to achieve better performance. For the very efficient CSL algorithm [CH90], this is done in the following section.

3 Extending the CSL Algorithm

Before we will specify our extension, we shortly recall the original distributed mutual exclusion algorithm of Chang, Singhal and Liu (CSL) [CH90]. This algorithm is a dynamic, logical structure based and token based algorithm. For an extensive discussion of other classification issues, see [SI93]. It uses an efficient path compression technique [BE89] to reduce the height of the communication tree maintained at run-time. Its proven message complexity per critical section entry is O(log N), where N is the number of participating processors. This algorithm performs excellently under various load conditions [CH96].

This short description of the CSL algorithm is mainly taken from [JO94]. The key idea of this algorithm is that each node maintains a *guess* about which node actually holds the token in a local variable called `root`. If a node neither holding nor requesting the token receives a write request, it forwards this write request to the node indicated by `root`, and then sets `root` to the number of the requesting node (since it will be the one which probably owns the token at the time of the next request).

When a node requests the token, it sends a `request` message to the processor indicated by `root`. It then sets an additional pointer, `next` to `NIL`. If a node that holds or is waiting for the token receives a request, and its `next` pointer is `NIL`, it sets `next` to the number of the node that sent the request. Otherwise, it forwards the request to the node indicated by `root`, and sets `root` to the requesting node. Among the nodes that hold the token or are requesting or idling, the `next` variable form a distributed queue of the blocked processes. If a process is waiting and its `next` pointer is `NIL`, the process is effectively at the end of the waiting queue. If `next` is not `NIL`, the end of the waiting queue is at the node pointed to by `root`, or even beyond. Since the requesting process will become the one at the end of the list, it is appropriate to set the `root` variable to the number of the requesting process. When the token holder releases the token, it sends the token to the node pointed to by `next`, if `next` is not `NIL`. Otherwise, the token holder keeps the token without using it.

The original algorithm supports pure mutual exclusion functionality only. Our extension adds distributed single writer / multiple reader semantics as introduced above to the original algorithm.

3.1 Data Structures local to each Node

Like in the CSL algorithm, each node participating in the new algorithm has to maintain local data structures for each process on the node which is synchronizing. Each process may maintain an arbitrary number of tokens, which are represented with appropriate C++ style lock objects used for the description. In our implementation, these data structures are maintained in an entity called token server daemon (TSD). In practice, this daemon can efficiently be mapped, for instance, onto a classical UNIX network daemon or a Windows NT network service. The TSD at each processor is structured as a set of C++ objects maintaining the data structures representing the state of a specific distributed reader / writer lock. The class interface for these objects look partly like:

```
class RWCSL : public _Object {
    ...
    protected:
        int lockId;               // global lock id
        int root;                 // root of the directed tree
        int first, next;          // first and next requester
        int state;                // request state of the node
        Queue *sharedRequestQueue;// local shared request queue
        int numCopies;            // counter for token copies
        int copyManager;          // manager of the current copies
};
```

The TSD‘s representation of a lock object in user space defines the identity of the lock owner, the global unique identifier for the lock object and the request state of the lock object.
In more detail, the `lockId` instance variable introduces a global name scheme to allow an arbitrary number of lock objects which are represented by a set of `RWCSL` instances within each TSD. The `root` and `next` instance variables have the same semantics as in the original

CSL algorithm. The first variable, however, is an extension to the CSL algorithm for the special case where a local exclusive request for the token cannot be granted because the current number of copies of the token is not zero. In this case, the local node is the first and the last requesting node in the distributed waiting queue (this case cannot happen in the original algorithm).

The request state of the local processor is stored in the state instance variable, which may have one of the values Clear (node is idle), RequestingRead (node is requesting for shared ownership), Requesting (node is requesting for exclusive ownership), ConsumingRead (node is in critical section for reading), Consuming (node is in critical section for writing) and HoldingFreeToken (node owns the token but is not requesting it.).

The basic idea of our extension is to maintain local queues of pending shared requests along the distributed waiting queue of current exclusive requests to the token. The token may leave a node only if the local queue of shared requests (sharedRequestQueue) is empty and the number of copies is zero.

3.2 Initialization Phase and Message Types

Each RWCSL instance initializes such that a node determined by hashing the global lock id (function hashlockId()) becomes root of the directed tree. This node is holding the free token initially. The myid() function returns the local node number to the caller:

```
void RWCSL::initWith(int aLockid) {
     _Object::init();
     lockId = aLockId;
     root = hashLockId(aLockId);
     first = next = copyManager = NIL;
     state = (myid() != root) ? Clear : HoldingFreeToken;
     sharedRequestQueue = (new Queue)->init();
     numCopies = 0;
}
```

As in the original CSL algorithm, messages of type Token to grant a token to a remote TSD and of type Request to request a token from a remote TSD are dispatched. Additionally, we parametrize these two types of messages with the mode of the request (either exclusiveRequest or sharedRequest).

3.3 Handling exclusive Requests of a remote TSD

If the local number of copies is greater than zero and first points to NIL, nobody is requesting the token. Hence first is set to the sender sender of the message. If the local node is not equal to root, the request is forwarded. Otherwise, next must currently be NIL and can be used to store the request. If there are no shared copies of the token but the local node is already requesting or consuming the token, the request is handled like in the standard CSL algorithm. That is, it is stored in the next variable if the local node is the last requesting node in the distributed queue or it is forwarded if it is not. If the local node is holding the free token, it can be granted to the requesting node immediately. If it is not, the request is also forwarded, respectively. Finally, root is set to the number of the requester, namely sender.

```
void RWCSL::exclusiveRequest_Handler(Message *msg, int sender) {
    if (numCopies > 0) {
        if (first == NIL) {
            first = sender;
        } else if (root != myid()) {
            sendMessageTo(root, msg);
        } else {
            next = sender;
        }
    } if ((state == Requesting) || state == Consuming)) {
        if (next == NIL)
            next = sender;
        else
            sendMessageTo(root, msg);
    } else if (state == HoldingFreeToken) {
        next = NIL; root = sender;
        sendMessageTo(sender, prepareMessage("exclusiveToken"));
        state = Clear;
    } else
        sendMessageTo(root, msg);
    root = sender;
}
```

3.4 Handling shared Requests of a remote TSD

Requesting the token for shared ownership is implemented as follows. If the local node is not equal to `root`, the request is forwarded. Otherwise, a copy of the token can be granted, if the local node itself owns a copy or if it is even holder of the free token. In the other case, the node must be requesting or consuming exclusively and hence, the request must be enqueued locally:

```
void RWCSL::sharedRequest_Handler(Message *msg, int sender) {
    if (root != myid())
        sendMessageTo(root, msg);
    else {
        if((state==ConsumingRead)||(state==HoldingFreeToken)) {
            numCopies++;
            sendMessageTo(sender, prepareMessage("sharedToken"));
        else
            sharedRequestQueue->enqueue(sender);
    }
}
```

3.5 Receiving the Token from another TSD

Again, while releasing the token the new shared mode must be dispatched. For exclusive releases, the code is basically the same as in the original CSL algorithm: If the local node number is different from the current `root` of the sender (`msg->actRoot`), the local `root` value is set to the new value. The state must then be set to `Consuming`. However, we found that applying the path compression technique in this case is prone to deadlock (even in the unmodified CSL algorithm!). The deadlock danger arises from the fact that message delays are unpredictable. We found that, even in the original algorithm, too early token messages may destroy the distributed list of nodes connected by the next pointers. A workaround is *not* to modify the root value when exclusive tokens are received.

```
void RWCSL::exclusiveToken_Handler(Message *msg) {
     if (msg->actRoot != myid())
          root = msg->actRoot; // dangerous!
     state = Consuming;
}
```

If a TSD is receiving a shared token, it acts as follows:

```
void RWCSL::sharedToken_Handler(Message *msg, int sender) {
     copyManager = sender;
     if (msg->actRoot != myid()) // should always be true
          root = msg->actRoot;
     state = ConsumingRead;
}
```

The sender of the message `sender` must be the current manager of the copies of the token, hence its value is stored in the local `copyManager` variable. Note that the path compression technique also applies to this case: `root` can be set to the current root of the sender, which may reduce the height of the directed tree significantly. At last the state is set to `ConsumingRead`.

3.6 Communication with user processes

We will now present the code actually requesting or a releasing a token by a user process (in shared or exclusive mode).

3.6.1 Exclusive Request / Release

If a user process wants to enter a critical section exclusively, it calls the following method:

```
void RWCSL::doExclusiveRequest() {
     if ((state == HoldingFreeToken) && (numCopies == 0)) {
          state = Consuming;
          return;
     }
     if (root != myid())
          sendMessageTo(root, prepareMessage("exclusiveRequest"));
     else
          first = myid();
     state = Requesting;
     root = myid();
}
```

If a user process wants the token for exclusive ownership and the local TSD is holding a free token and the number of token copies is zero, the request can immediately be granted. Otherwise, the message is forwarded to `root`. If `root` is set to the own node number, the request must be stored in `first`.
The code to release the exclusive ownership of the token at the end of the critical section is:

```
void RWCSL::doExclusiveRelease() {
     if (sharedRequestQueue->isEmpty()) {
          if (next == NIL) {
               state = HoldingFreeToken;
               return;
          }
          sendMessageTo(next, prepareMessage("exclusiveToken"));
          next = NIL;
          state = Clear;
     } else {
          first = next; next = NIL;
          while(!sharedRequestQueue->isEmpty()) {
               sendMessageTo(sharedRequestQueue->dequeue(),
                              prepareMessage("sharedToken"));
               numCopies++;
          }
          state = HoldingFreeToken;
     }
}
```

If the queue of pending shared requests is empty and `next` is `NIL`, there are no pending requests at the node, hence state is set to `HoldingFreeToken`. Otherwise, the exclusive token is granted to the next requesting process. If there is a pending shared request, `first` is set to `next` and copies of the token are granted to all requesters in the `sharedRequest-Queue` queue.

3.6.2 Shared Request / Release

If a user process wants to enter a critical section for reading only, it calls the following method:

```
void RWCSL::doSharedRequest() {
     if ((state == HoldingFreeToken) && (root == myid())) {
          numCopies++;
          copyManager = myid();
          state = ConsumingRead;
          return;
     }
     sendMessageTo(root, prepareMessage("sharedRequest"));
     state = RequestingRead;
}
```

If the node is holding a free token and the local node is currently `root`, a copy of the token can be granted to the requesting process immediately. Otherwise, the request is forwarded to `root`.

To leave the critical section, the following method is called:

```
void RWCSL::doSharedRelease() {
     state = Clear;
     if (copyManager == myid()) {      // local shared release
          state = HoldingFreeToken;
          releaseToken_Handler(prepareMessage("localRelease"))
     } else                            // remote shared release
          sendMessageTo(copyManager, prepareMessage("releaseToken"));
}
```

... where the **`releaseToken`** message handler is called by the node `copyManager` and dispatches only shared releases of the token by:

```
void RWCSL::releaseToken_Handler(Message *msg) {
     numCopies--;
     if (numCopies != 0)
          return;
     if (first == myid()) {            // I am the next to get the token
          S = Consuming;
          first = NIL;
     } else if (first == NIL) {        // nobody needs the token
          state = HoldingFreeToken;
     } else {                          // someone requested the token
          sendMessageTo(first, prepareMessage("exclusiveToken");
          first = NIL;
          if (state == HoldingFreeToken)
               state = Clear;
     }
}
```

Like denoted in the introduction, a special case occurs when a local requester is blocked due to the fact that the token is blocked by a non-zero `numCopies` value. This special case cannot happen in the original algorithm, but is handled carefully in our extension.

4 Evaluation of the Extension

In this section, we evaluate the extended algorithm by integrating it into our forthcoming all-software DSM system providing the shared virtual memory (SVM) abstraction [PA97] to parallel applications.

4.1 Overview of SVMlib

SVMlib (*Shared Virtual Memory Library*) has been designed to benefit from several features provided by Windows NT, like preemptive multithreading and support for SMP machines. Unlike most software DSM systems, the SVMlib itself is truly multithreaded. SVMlib also supports multithreaded user-code to take advantage of clustered Windows NT SMP systems.

SVMlib currently runs on a cluster of Intel Pentium Pro 200 Dualprocessors connected by FastEthernet via TCP/IP. SVMlib will also support efficient message passing using Dolphin‘s implementation of the Scalable Coherent Interface [IE92]. The library also runs on Solaris 2.5.1, either SPARC or Intel.

The library provides the notion of distributed shared memory by providing *virtually shared memory regions*, an approach widely used in other SVM implementations [BE95]. Page faults within virtually shared memory pages are handled at user-level via structured exception handling provided by the C++ run-time system of Windows NT. At the current stage, SVMlib supports two important memory consistency models: the widely-used, though fairly inefficient sequential consistency and an all-software implementation of the lock associated scope consistency [IF96]. The latter was chosen because of the tight integration with distributed synchronization primitives.

4.2 Integration of the extended Distributed Synchronization Algorithm into SVMlib

SVMlib makes extensive use of the proposed extended distributed synchronization algorithm at different levels. First of all, it is directly being used at user-level to offer distributed reader/writer locks to parallel application programs. The second level is the SVM layer itself, where distributed synchronization is indirectly necessary to maintain consistency of the virtual shared memory region managed.

- To support *sequential consistency*, the extended distributed synchronization algorithm offers direct support for concurrent readers on virtually shared pages. This is fairly easy achieved by associating the contents of a virtually shared page with a token handled by the extended distributed synchronization algorithm;
- Protocols providing *relaxed consistency* associate consistency update information with distributed synchronization primitives like distributed exclusive locks. This holds for the classical *lazy release consistency* [KE92], as well as the fairly new *scope consistency* [IF96]. Our extended algorithm allows an improved implementation of these consistency models where update information is only generated and sent if a lock had been acquired by the sender in exclusive mode. Thus, message traffic related to the propagation of consistency information can be reduced if a portion of the participating processes acquires the locks in read-only manner.

In the following subsection, we evaluate the performance of the modified algorithm in a event-driven simulation. We also present real performance results within the UNIX version of SVMlib.

4.2.1 Performance Measurements of the Algorithm

Simulation

We compared the efficiency of the original CSL algorithm with our extension in terms of message and time complexity for different mixes (shared / exclusive) of 100.000 token requests. For our simulations, we assumed fixed cost for message sending, while both, token request arrival times from user processes and the lengths of the critical sections are exponentially distributed. To compare with the original algorithm, we measured four different cases, ranging from 0% read request percentage (e.g., *every* request is for exclusive ownership, like in the original CSL algorithm) up to 80% read request percentage. Fig. 2 shows the message complexity on a simulated system with up to 64 nodes for 0%, 30%, 50% and 80% read requests:

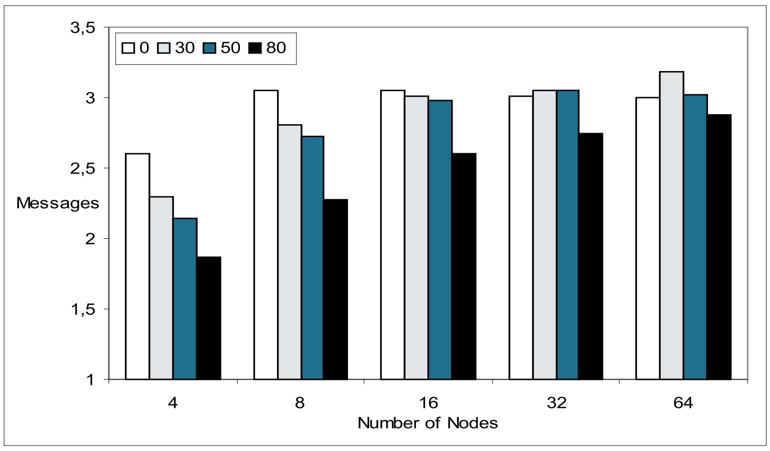

Fig. 2: Acquire Message Complexity

In the second benchmark, the mean virtual simulation time to acquire a token is measured (either in shared or in exclusive mode, depending on the shared request percentage, ranging from 0% to 80% of all requests):

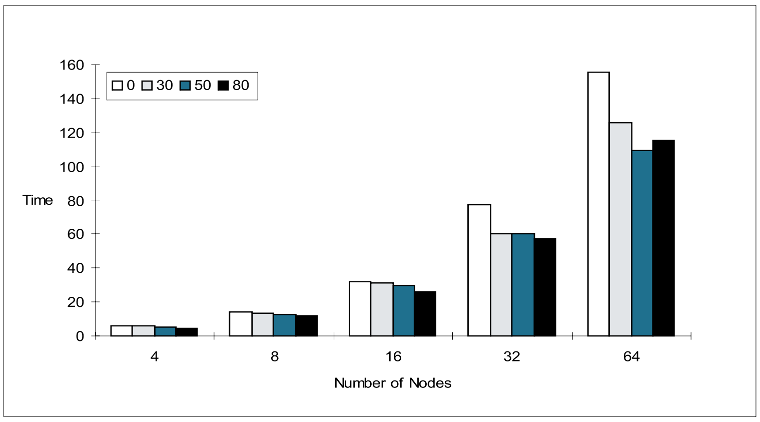

Fig. 3: Acquire Response Times

Finally, we measured more different read request percentages (ranging from 0% to 100%) on a fixed moderately sized simulated system with 16 nodes. The figure shows the message complexities for shared requests („Req-Read“), shared release messages („Rel-Read“) compared with an CSL-like exclusive access („Write Access“):

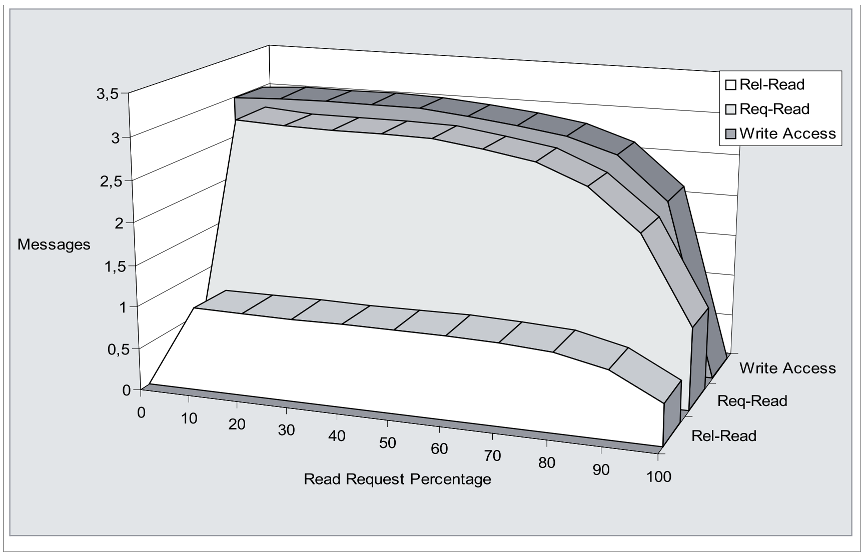

Fig. 4: Acquire/Release Message Complexity (16 Nodes)

These benchmarks show that, up to 64 nodes, the message complexity to acquire a token in "increasingly" shared mode percentage is significantly more efficient than using the unmodified CSL algorithm for exclusive requests. For example, for 8 nodes and 80% read request percentage, the modified protocol needs to send about 2.3 messages on the average, compared to slightly more than 3 messages for unmodified CSL. The message complexity for 0% read request percentage is identical to CSL. There is, however, a scaling problem with our extension, when the number of nodes is really big (e. g., more than hundred processors). This is due to the fact that, at high read request percentages, the queues of pending read requests grow linearly, with an increasing impact on the response times. In this case, on the other hand, the drawback is complemented by the fact that our protocol extension allows a significantly higher lock utilization by allowing parallel processes to enter their critical sections concurrently, while original CSL does not.

Performance within SVMlib

Besides the above event driven simulation, we now present the „real-world" performance of the implementation of the modified algorithm within our SVMlib library. We compare our algorithm (RWCSL) with the distributed reader / writer algorithm of Johnson et al. [JO96].
The results in Figure 4 where made using the Solaris 2.5.1 version of SVMlib, since our PC cluster is not big enough at the moment. For the experiment, 8 SPARC SS-20 connected by 10 MBit/s EtherNet via TCP/IP were taken. The graphics shows two things:

- The *mean total protocol overhead times* (*RWCSL Total*) when acquiring and releasing a token in increasingly shared mode compared to the algorithm of [JO96] (*JO96 Total*). To measure the pure overhead, the critical sections were empty.
- The *mean token acquire times* (RWCSL Acq) when acquiring a token in increasingly shared mode compared to the algorithm of [JO96] (*JO96 Acq*).

.

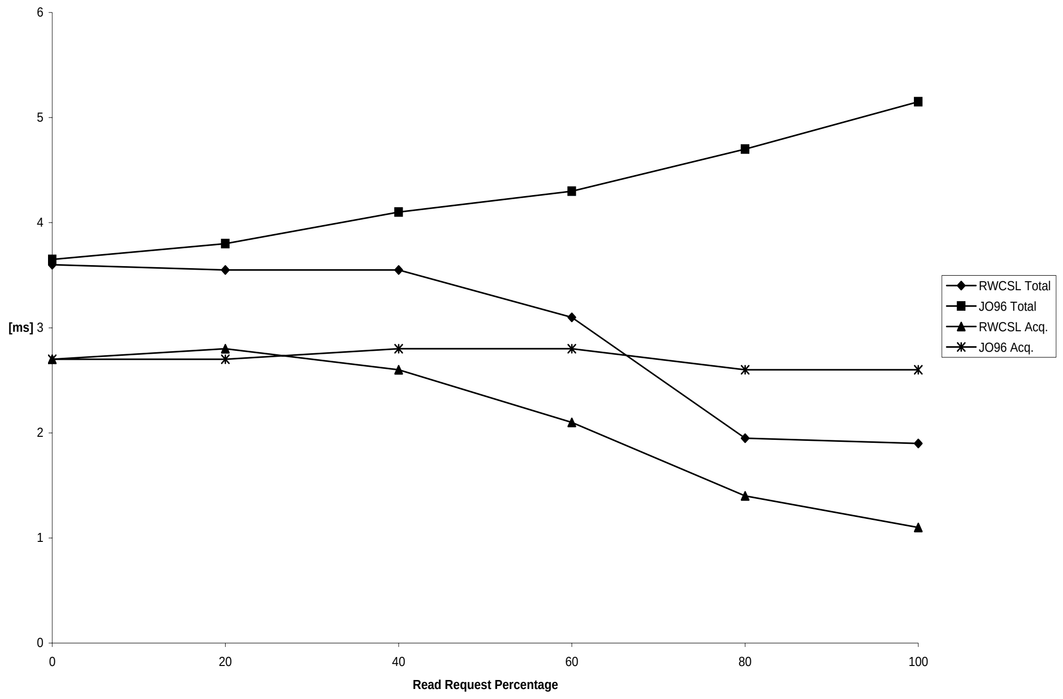

Fig. 4: Performance within SVMlib (Solaris 2.5, SPARC SS-20, EtherNet, 8 Nodes)

5 Summary

We have presented an extension to an existing efficient algorithm for distributed mutual exclusion. We showed how to extend the notion of single reader / multiple writer locks to distributed environments and how to integrate this semantics in a practical and easy to implement way into the path compression technique of Chang, Singhal and Liu [CH90]. We evaluated our new extension by simulation and by experiments with a newly developed all-software, page-based DSM system for clustered Windows NT systems. We found that our extension increases the lock throughput and lock utilization significantly while preserving fairness between mixed requests and no additional message complexity for exclusive requests. Future work includes benchmarking the performance of our extension by non-synthetic (e.g. real world application) synchronization loads.

References

[BE89] J. M. Bernabeu, M. Ahamad: *Applying a path-compression technique to obtain an effective distributed mutual exclusion algorithm*, Proc. of the 3rd International Workshop on Distributed Algorithms, Nice, France, 1989

[BE95] Berrendorf, R.; Gerndt, M.; Mairandres, M.; Zeisset, S.: *A Programming Environment for Shared Virtual Memory on the Intel Paragon Supercomputer*, ISUG Conference, Albuquerque, 1995

[CH90] Y.-I. Chang, M. Singhal, M. T. Liu: *An improved O(log n) mutual exclusion algorithm for distributed systems*, International Conference on Parallel Processing, pp. III 295-302, 1990

[CH96] Y.-I. Chang: *A Simulation Study of Distributed Mutual Exclusion*, in: Journal of Parallel and Distributed Computing, Vol. 33, No. 2, March 15, 1996

[CH94] D.-K. Chen, H.-M. Su, P.-C. Yew: *The Impact of Synchronization and Granularity on Parallel Systems*, Center for Supercomputing Research and Development (CSRD), University of Illinois at Urbana-Champaign (UIUC), Technical Report TR #942, 1994

[DI89] A. Dinning: *A Survey of Synchronization Methods for Parallel Computers*, IEEE Computer, pp. 66-77, July 1989

[IE92] IEEE: *ANSI/IEEE Std. 1596-1992, Scalable Coherent Interface (SCI).* 1992.

[IF96] L. Iftode, J. P. Singh, K. Li: *Scope Consistency: A Bridge between Release Consistency and Entry Consistency.* 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA‘96), June 1996

[JO94] T. Johnson: *A Performance Comparison of Fast Distributed Synchronization Algorithms*, Univ. of Florida, Dept. of CISE, Technical Report TR #94-032, 1994

[JO96] T. Johnson: *A Fair Fast Distributed Concurrent-Reader Exclusive Writer-Synchronizatio*n, Univ. of Florida, Dept. of CISE, Technical Report TR #96, 1996

[KE92] Keleher, P.; Cox, A. L.; Zwaenepoel, W.: *Lazy Release Consistency for Software Distributed Shared Memory.* Proc. of the 19th Int. Symp. on Computer Architecture (ISCA‘92), pp 13-21, 1992

[KR93] O. Krieger, M. Stumm, R. Unrau, J. Hanna: *A Fair Fast Scalable Reader-Writer Lock*, Proc. International Conference on Parallel Processing, 1993

[LA78] L. Lamport: *Time, clocks and the ordering of events in a distributed system*, Communications of the ACM, Vol. 21, No. 7, pp. 558-565, July 1978

[MA85] M. Maekawa: *A sqrt(n) Algorithm for Mutual Exclusion in Decentralized Systems*, ACM Transactions on Computer Systems, Vol. 3, No. 2, pp. 145-159, May 1985

[MA92] K. Makki, K. Been, P. Banta, N. Pissinou: *On algorithms for mutual exclusion in distributed systems*, International Conference on Parallel Processing, pp. II 149-152, 1992

[MJ85] A. J. Martin, *Distributed Mutual Exclusion on a Ring of Processors*, Scientific Computer Programming 5, 1985

[NA87] M. Naimi, M. Trehel: *An improvement of the log(n) distributed algorithm for mutual exclusion*, 7th International Conference on Distributed Computing, pp. 371-375, 1987

[PA97] S. M. Paas, M. Dormanns, K. Scholtyssik, S. Lankes: Computing on a Cluster of PCs: Project Overview and Early Experiences, 1st Workshop on Cluster-Computing, TU Chemnitz-Zwickau, 1997

[RA94] M. Ramachandran, M. Singhal: *On the Synchronization Mechanisms in Distributed Shared Memory Systems*, Technical Report OSU-CISRC-10/94-TR54, 1994

[RA89] K. Raymond: *A Tree-Based Algorithm for Distributed Mutual Exclusion*, ACM Transactions on Computer Systems, Vol. 7, No. 1, pp. 61-77, 1989

[SI94] A. Silberschatz, P. Galvin: *Synchronization in Solaris 2*, in: *Operating System Concepts*, 4th edition, Addison-Wesley, 1994, pp. 198-199

[SI93] M. Singhal: *A Taxonomy of Distributed Mutual Exclusion*, Journal of Parallel and Distributed Computing 18, pp. 94-101, 1993

[SR92] P. K. Srimani, S.R. Das: *Distributed Mutual Exclusion Algorithms*, IEEE Comp Society Press, 1992

[SU85] I. Suzuki, T. Kasami: *A distributed mutal exclusion algorithm*, ACM Transaction on Computer Systems, Vol. 3, No. 4, pp. 344-349, 1985